Michał Śmiałek

Software Development with Reusable Requirements-Based Cases

November 13, 2007

Warsaw University of Technology

Michał Śmiałek

Instytut Elektrotechniki Teoretycznej i Systemów Informacyjno - Pomiarowych

SOFTWARE DEVELOPMENT WITH REUSABLE REQUIREMENTS-BASED CASES

Manuscript submitted 24.10.2007

Vast majority of software development projects seem to ignore past knowledge about solving specific problems. This might be explained by significant difficulties to reuse knowledge in such a complex domain as software engineering is. There seem to be no effective mechanisms to find and reuse past solutions to problems similar to the currently solved one. The main question that this book aims at solving is the above inability to reuse knowledge about already solved software development problems. In this book there is proposed a process for systematic reuse of so-called Software Cases. Any Software Case contains a precisely expressed problem statement in the form of a Requirements Model. All elements of this problem statement can be mapped onto appropriate elements of the problem solution. This solution is formed of (again) precisely expressed design models and the final code. Software Cases can be reused on the basis of their similarity to the currently developed system (current Software Case). This similarity can be determined by comparing the current (perhaps yet incomplete) requirements model with requirements models of past Software Cases. The past solution can then be easily reused by modifying it in those places that are precisely marked as needing rework in order to solve the current problem.

The book contains a detailed discussion on the issues that lead to constructing a comprehensive requirements-based reuse framework. There are described mechanisms and tools that can support such a framework. A vision of how to organise a reuse process is presented, including details on how a software development organisation should follow this process. This includes using a specific precise language for specifying requirements and designing systems. The process and the language are defined both formally and practically. The book introduces concrete syntax for individual elements of Software Cases: requirements, architecture and detailed design. This syntax is used to formulate them in a systematic way. Techniques for transform-

ing models to form a coherent path from requirements to code are given. Certain mechanisms for comparing and retrieving Software Cases are also specified. This includes a query language suitable for formulating queries that allow for matching requirements models thus allowing for reusing solutions to problems specified through requirements.

Key words: software reuse, software requirements, software model transformations, software development methodologies

1

Introduction and rationale

1.1 Problems faced by software development projects

Contemporary software systems become more and more complex. The complexity of these systems is associated with the complexity and changeability of problems specified through requirements specifications. Complex problems lead to even more complex solutions implemented in the technological space that changes even faster than the problem domain. Despite this complexity (or maybe - due to it), vast majority of software development projects seem to ignore past knowledge about solving specific problems. This might be explained by significant difficulties to reuse knowledge in such a complex domain as software engineering is. The main obstacle here is that there is no standard way to capture knowledge about complete cases leading from the problem (requirements) to its solution (design and code). There also seem to be no effective mechanisms to find and reuse past solutions to problems similar to the currently solved one.

The main question that this book aims at solving is the above inability to reuse knowledge about already solved software development problems. This question is illustrated in Fig. 1.1. A software development (SD) project produces specific artifacts (requirements documents, design documents, code, etc.). Additionally, certain tacit knowledge is gained by the SD project participants. Some of the SD project results can be generalised to form design or analytical patterns. Unfortunately, the above mentioned artifacts are usually very hard to reuse, even when the new problem is very similar to the previous one (or is simply a next version of an existing system). This is caused by the fact that this knowledge is not structured in a way that would allow for easy comparison and retrieval. Even the use of design patterns is limited as it necessitates from developers certain thorough knowledge of pattern libraries that go beyond the 23 classical patterns found in [70]. Moreover, in order to prepare reusable assets or patterns, significant effort is needed in advance, without clear perspectives of actual return on that effort. We thus need effective mechanisms for formulating software knowledge, finding simi-

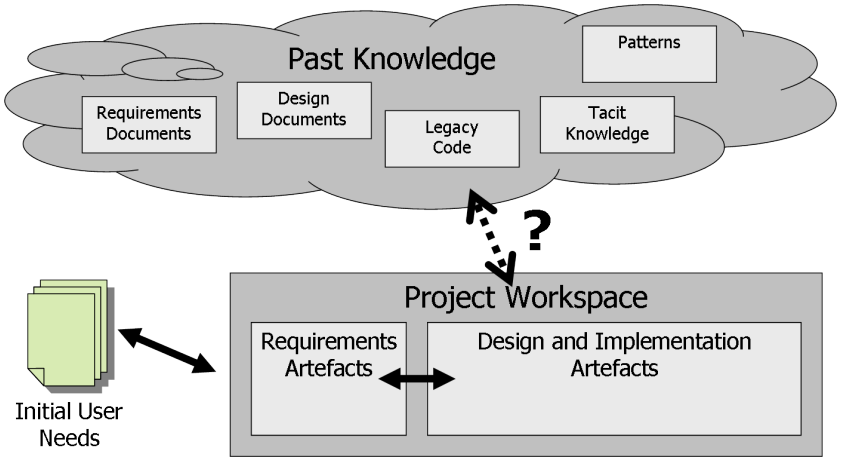

Fig. 1.1. Reusing unstructured knowledge in software development projects

larities between problem specifications (requirements) and then for showing appropriate differences in the solutions to these problems, to be re-developed in the new SD project. These mechanisms should lead to significant reduction in effort when preparing for reuse and when performing reuse thus reducing its cost and encouraging software development organisations to apply reuse-oriented methods in their everyday practice.

1.2 Main concept behind this book

The above "amnesia" problem is not tackled properly by the current approaches to software reuse, which require significant effort for introducing reuse mechanisms into organisations and then - for preparing software artifacts (structuring software knowledge) for reuse and seeking for that knowledge. In order to reduce this "amnesia" we need to reduce the barriers for reuse by introducing easy to apply and thus inexpensive mechanisms. We need a complete framework for systematic creation and reuse of software development artifacts in all disciplines of software engineering (including requirements, design and implementation). This framework would thus facilitate reuse of complete software development cases that comprise the problem statement (requirements) and problem solution (design and implementation). These cases would need to be organized not as a set of loosely coupled documents, models or code. Models in such reusable cases should be precisely and tightly mapped one onto another and then - onto code.

The main objective of this book would thus be to propose a process for systematic reuse of software cases, as illustrated in Fig. 1.2. Such a Software Case contains a precisely expressed problem statement in the form of a Re-

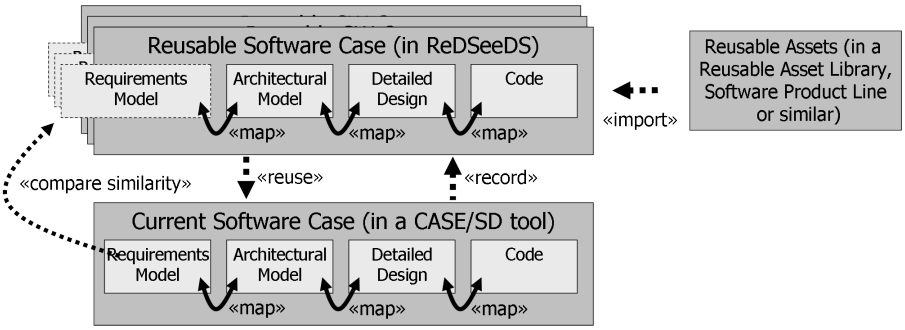

Fig. 1.2. Requirements-based reuse of structured software knowledge

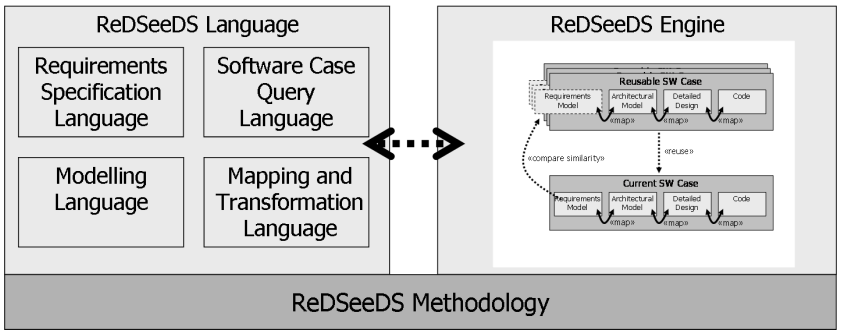

Fig. 1.3. Elements of the ReDSeeDS Framework

quirements Model. All elements of this problem statement are then mapped onto appropriate elements of the problem solution. This solution is formed of (again) precisely expressed design models (including Architectural Model, Design Model, etc.) and the final code. Software Cases can be reused on the basis of their similarity to the currently developed system (current Software Case). This similarity can be determined by «comparing» the current (perhaps yet incomplete) requirements model with requirements models of past Software Cases. The past solution can then be easily «reused» by modifying it in those places that are precisely marked as needing rework in order to solve the current problem.

It can be noted that a very important feature of such an approach is "seamlessness" in formulating reusable cases. The current Software Cases will have exactly the same structure as the past cases. This means that the only effort associated with making the current case reusable («recording» it) is associated with copying it directly into the reuse repository. Of course, certain effort will be needed to «import» from the existing reusable asset libraries, software product lines and similar.

In order to apply the above mechanism in practice, we will propose in this book a comprehensive reuse-oriented, requirements-based software development framework. We will call this framework a Requirements Driven Software Development System (or shortly: ReDSeeDS). This ReDSeeDS Framework, illustrated in Fig. 1.3, would be composed of three elements: ReDSeeDS Language, ReDSeeDS Engine and ReDSeeDS Methodology. The purpose of the ReDSeeDS Language would be to allow for precise representation of reusable Software Cases. The language will be composed of a Requirements Specification Language (RSL), a Modelling Language (ML), a Mapping and Transformation Language (MTL) and a SW Case Query Language (SCQL). The first three languages serve to define cases. The last language is used to find similar cases and mark differences between them. The ReDSeeDS Engine enables software developers to use the ReDSeeDS Language and realize the reuse mechanism from Fig. 1.2 in practice. Finally, the ReDSeeDS Methodology describes the software lifecycle based on creating and reusing SW Cases defined with the ReDSeeDS Language and performed through the ReDSeeDS Engine. The methodology aims at receiving significant levels of reuse with the developed engine. At the same time, these levels would be accomplished with a significantly lower effort than it is possible today. There would be no need to prepare for reuse by creating special "manifestation artifacts", laboriously generalize components, or conduct complex "variability analysis".

1.3 Current state of the art

In order to meet the above objectives we need to combine several areas of research in software engineering. The ReDSeeDS framework aims at systematic reuse of software and thus we need to build on the state of the art in reusable asset libraries (including software product lines). The assets reused from the library will be mostly models (graphical and textual) linked through mapping or transformation relationships. In this area, we need to apply the state of the art in model-driven development and transformations (including graph transformations). Model and graph technologies will also be used to develop query mechanisms for reusing models. These would need to be combined with state of the art in reasoning which is also used for retrieving (reusing) knowledge. Moreover, these reuse mechanisms will be coupled with the state of the art in requirements engineering to enable requirements-based reuse. This last area will also be combined with model-driven development to allow for case-based reuse of models mapped from requirements.

1.3.1 Model-driven development and transformations

Complexity of contemporary software systems is constantly growing. In order to deal with this complexity effectively, appropriate abstraction techniques are needed. Creating models of software, slowly becomes the most prominent

method for dealing with this complexity and realizing abstraction. Models allow for managing the structure and dynamics of even the most complex systems on different levels of abstraction, making it possible to reveal only the amount of detail that is comprehensible for the model readers.

Here we will concentrate on modeling of software systems based on the widely used object oriented paradigm. After an initial boom in the area of software modeling methodologies based on object orientation (see eg. [133, 226, 225, 37, 38, 30, 177, 97, 145, 81, 90, 187, 91, 83]), current tendency is to unify different approaches (see [178]). The most prominent example in this area is the Unified Modeling Language (UML) [153, 154] currently in its version 2.0. UML is already an ISO standard (in version 1.4 [96]) used by vast majority of companies in the software industry and managed by an independent standardization body - the Object Management Group (OMG, www.omg.org). UML in its current version offers thirteen diagram types that enable showing various aspects of software and associated problem (business) domains on different levels of abstraction. Numerous handbooks on UML exist [43, 160, 173, 68, 193, 123, 7, 163]. Current development of the UML standard goes in the direction of adding precise semantics to its diagram elements. Defining precise semantics allows for bringing closer the idea of "visual programming", where coding would mean drawing diagrams instead of writing textual code ("executable UML" - see [127, 168]).

Beyond UML, there exists an interesting, open extension called SysML[1] [205] which is intended to be a systems modelling language, where a system may include hardware, software, information, processes and even personnel. In this sense, SysML extends UML from software to more general systems. It is interesting to note in the context of this book that SysML explicitly includes requirements as modelling elements. There is also a strong research and application area of so-called domain-specific modelling languages [210]. These languages are defined for specific application domains by the software developers themselves. It can be argued that such languages have an advantage over UML of having concrete notation more readable by the language users. However, counter arguments include the amount of effort needed to develop a specific language for a specific purpose and effort needed to learn the languages' syntax and semantics. Finally, it is worth noting that there exist also "variability modelling" languages oriented on reuse issues. An example of such language is AMPL (Asset Modelling for Product Lines) defined with the meta-modelling language AMPL-M [93].

When modeling software we have to remember that the overall model consists of strongly interweaved and interconnected models, describing requirements, software architectures, as well as executable programs. The most significant problem in using UML is to organise diagrams into models that would form a clear path from user requirements, through architectural design to detailed design and code. Taking this into account, the OMG has

[1] http://www.sysml.org/

introduced a new modelling framework, called Model Driven Architecture (MDA) [138, 111, 128]. MDA can be viewed as the currently most important and influential vision of model-based software development. Software development based on MDA uses modelling and meta-modelling techniques and also extends them with the concept of model transformation. According to this idea, models close to the business domain (Computation Independent Models) would get transformed into abstract architectural models of the system, independent of the programming platform (Platform Independent Models). These models could then be transformed into models dependent on the software platform (Platform-Specific Models), and then directly translated into working code. Such a clear path, from visual, model based requirements, through design models to code could be partially automated, thus giving faster development cycles (see eg. [21], [156]), although certainly does not yet constitute a panacea for all the software engineering problems [209].

Models used in MDA are expressed in a modelling language combining (as every formal language) three elements: abstract syntax (language schema), concrete syntax (notation visible to the users) and semantics (meaning of the language constructs). The concrete syntax of textual languages is typically defined by context free grammars, while abstract syntax, especially for graphical languages is defined by means of meta-models (typically in the MOF [155] notation). MOF 2.0 (which has a common infrastructure with UML 2.0) has already become a widely accepted notation for meta-modeling.

Next to modelling, model transformation [23] plays an important role in all model-based approaches. Transformations convert models (starting from requirements models [196]) into further models including executable code (see eg. [126] for an interesting approach). Requirements for transformation languages within the MOF meta-modeling environment are proposed by OMG in the Query/View/Transform proposal [147]. Various approaches to fulfill these requirements have been constructed by industry dominated consortia (e.g. [167, 166]). QVT-Merge [166] is an initiative of QVT-Merge Group, involving most of the active players in the area. Therefore this proposal most likely will become the basis of the new standard. The proposed language has both textual and graphical forms, and is based on the use of a relatively simple pattern mechanism for defining transformations. Slightly extended OCL (Object Constraint Language) 2.0 [152] is used for model queries. Several problems with the QVT-Merge approach mean that independent research groups are working on other approaches (see e.g. [112], [56], [104]).

Transformation approaches can be divided into graph- or text-oriented variants, which are both relevant here. A coarse overview on transformation techniques is presented in [51] and a first sketch of a comparison of graph- and text-based transformation techniques [217] was presented. Textual transformation techniques based on term rewriting include ASF+SDF [213], Stratego [216] or TXL [47]. These approaches are successfully applied within program transformation, based on suitable grammars for the appropriate languages. An overview on program transformation approaches is given in [215]. Further

approaches contain template languages like XSLT [218] or TDL [10]. These approaches are based upon patterns, which are detected during traversal of models and transformed into a target structure. Graph-based transformation techniques are based on various underlying graph models. Representatives of these approaches are AGG [207], ATL [22], BOTL [31], FUJABA [211, 230], MOLA [104], and PROGRES/MDI [183, 113]. Transformation rules in these approaches contain patterns of a source graph, whose occurrences are transformed according to a target pattern. An interesting approach based on graph-rewriting in the context of model reuse was given in [119]. [129] present a taxonomy of transformation systems and compare some graph-based transformation techniques.

Despite the complexity of problems associated with model transformation, the purpose is to reduce efforts of software developers. Proper transformations applied in a software development project can make it very agile as it was discussed in [191].

1.3.2 Precise modelling of requirements

An important area in general software modelling is requirements modelling. Efforts in this area lead in the direction of precise requirements specification as opposed to traditional "common prose" requirements. This precision is certainly needed by the software developers who need very specific information on the system's scope. On the other hand, common prose requirements seem to be more acceptable by the users who want to understand them in the context of their everyday business. Unfortunately, it is a common observation that formal and thus precise requirements are hard to read and understand for the "ordinary people". On the other hand, requirements written in common prose, and acceptable for the users, are usually too ambiguous for the system architects and designers. It is widely discussed that eliciting requirements should involve as many diverse group of people as possible, thus improving the quality of the resulting system (see eg. [146] for a discussion).

It can be argued that the most important problem in requirements engineering is plethora of approaches to requirements specification. Various modelling notations have been introduced and used to create models of requirements. These notations include Entity Relationship Diagrams, UML Class Diagrams and Use Case Diagrams, SysML Requirements [89], Message Sequence Charts, Petri Nets, Data Flow Diagrams, Parnas Tables, State Machines, Decision Tables, Z, SDL and more. Generally these notations can be divided into formal (like Z) and informal or semi-formal (like UML). Formal notations have a precisely defined semantics with a textual or graphical syntax. Informal notations have their syntax defined although the notation has no clear interpretation or the interpretation of models depends on the current application. With formal notations, appropriate automatic transformations can be performed which facilitate tracing from requirements into design and test or code generation. On the other hand, informal notations can be seen as

much more readable by the users of business applications [45] which facilitates informal verification (see eg. [220] and [189] for a more detailed insight).

An interesting example of a dedicated formal requirements language is RML [86, 85]. This language introduces the concept of "conceptual models" where entities, activities and other real world phenomena are represented as objects in an object-oriented model.

The most prominent example of informal notation for requirements are Use Cases introduced around 1992 by Ivar Jacobson [97] and now present as an important element of UML. The use case notation together with domain class models can be seen as the currently most widely used model for requirements specification. This notation has certainly diminished the role of earlier notations (like ERDs and DFDs). Many handbooks on use cases exist (see eg. [40, 11, 182, 3, 117, 157]). This ubiquity of use cases can be seen as an excellent base for introducing a common requirements language to be used in reusable Software Cases as proposed in 1.2.

Unfortunately, since 1992 use cases have already been defined in so many ways by different authors that it certainly leads to huge confusion about the actual notation and specification techniques (see [13] for an insight). Alistair Cockburn back in 1997 has counted eighteen different definitions of use case [39]. These definitions differ in basically four dimensions: purpose, formality of contents, multiplicity of scenarios and formality of model. Depending on the definition, the use cases tend to be very formal or quite sketchy. Russel Hurlbut [94, 95] summarises almost 60 approaches towards use case specification. These approaches can be classified in terms of notation into textual, graphical and dynamic. Textual formats include unstructured text narratives, structured descriptions (templates), semi-structured scripts, formal expressions and tables. Graphical formats employ structure, state, interaction and implementation diagrams based often on the UML notation. Dynamic formats are based on animations and dynamic visualisations of the use case narration flow. Other notations include storyboards and role playing. This multitude of notations is caused mainly by imprecise definition of use cases by their inventor which was eventually not made more precise in the UML standard (including the latest version 2.0 [154]).

It can be argued that this ambivalence of use cases is caused by many targets that they aim at. Use cases are often utilised as the driving elements for the whole software development process. This was introduced by the use case inventor in [97] and then continued in various methodologies (see eg. [115, 174, 175]). The analysts write use cases to communicate their understanding of the prospective system's functionality. The users participate in formulating use cases to make sure their requirements are communicated well. The developers employ use cases to design architectures fulfilling the required functionality. User interface designers write storyboards based on use case scenarios. Testers design use case based test cases. If we browse different approaches summarised in [95] we can come to a conclusion that an "ideal" notation for use cases is impossible to reach. If the notation is to be general, it tends

to be rather informal in its nature. More formal notations are designed for specific purposes, like user interface specification [46], automatic requirements verification [200] or test automation [71]. Some use case notations impose architectural or design decisions (like the usage of UML interaction diagrams). Others use Use Case Charts [223], Petri Nets [52], activity diagrams [196] or BPMN diagrams [143] (for business process modelling). These last two approaches seem to be promising as they tend to combine formal definition of the flow of interactions with good comprehensiveness for the users not proficient in complex formal notations. Such graphical notations can be combined with textual approaches where scenarios [32] seem to play the most prominent role. However, under the term "scenario" there exist many notations (see eg. [82, 171, 172, 203, 5, 42, 101]), starting from simple structures (like in [24]) and ending with complex specification languages (as in [71]).

In the context of the idea presented in 1.2, an important issue is the ability to manage requirements models in tools. This includes tools that facilitate creation of use case based requirements models (see e.g. [57, 87, 197, 144]) by using the various notations mentioned above. An important element of such tools is facilitation of transition between the requirements models into design models (see [100] for a good discussion of such transition). The tools should allow for tracing the vast arrays of links between the requirement and design artefacts (see e.g. [59, 60] or [58]) and even try to perform automatic transformations of requirements into executable models (as proposed e.g. in [223]). It has to be noted that the currently available commercial requirements management tools like RequisitePro (by IBM) or DOORS (by Telelogic) offer only limited capabilities for tracing into design and concentrate more on management and traceability between paragraphs of text. The main problem here seems to be the lack of widely accepted detailed language or meta-model for precise requirements specification. This includes assuring consistency between functional and vocabulary requirements (as proposed eg. in [74] or [196]) together with non-functional requirements [49]. Unfortunately, attempts to define such a meta-model are sparse so far. Examples include approaches to requirements for Software Product Lines [14, 122] and general purpose requirements languages suitable for organizing mappings into design [102, 58, 60, 101, 191, 197].

1.3.3 Software reuse approaches

Most of the engineering disciplines are based on repeating certain patterns when designing and constructing solutions to specific engineering problems. An average engineer is usually taught these solutions during appropriate university courses. In disciplines like architecture or mechanical engineering, constructors reuse certain standard elements (eg. windows, doors, or ceilings in architecture) found in catalogs offered by various producers of such elements.

Also in software engineering, reuse as an idea is present practically since its emergence as a separate discipline [125]. However, is seems that despite

efforts and emergence of many reuse methodologies (see [53] for a survey, and [17] for an initial insight), this idea have failed so far [181] or causes tremendous problems to make it a success [139, 130]. It can be argued that this failure is caused by the breadth of problem domains where software systems are applicable. This makes it impossible to learn even the most important subset of standard reusable assets or even browse effectively through libraries containing them. Software development organisations wishing to implement software reuse in practice need to organise asset libraries specific for their domain areas (see [176] for survey of approaches). The stored assets need to be specially prepared in order to make it possible to reuse them in the future. This is a considerable investment with uncertain return on that investment necessitating quite complex ROI models [63, 135, 29].

An important specific problem encountered by software developers is the variety of ways in which reusable assets are stored and the variety of levels of reuse (see [136] for a good discussion and survey and [69] for a recent one). Reuse of code which is currently the most widely applied type of reuse, is usually limited to class libraries in the area of user interfaces and general data structures. Certain efforts to unify the structure of such reusable asset repositories are still not getting into the mainstream of software development. Despite this, certain important research activities in the area of methods and technology for engineering software reuse repositories have been undertaken [6]. The concept of reusable assets is accepted in the area of model-driven development. A standard has been introduced by the Object Management Group (OMG, www.omg.org) to unify the representation of software assets. This standard (Reusable Asset Specification [149]) supplies tool vendors with a unified meta-model for representing various artifacts produced throughout the software development and meant to manifest these artifacts in a reuse lifecycle.

Another stream of effort in applying reuse is the application of modelling patterns. These patterns offer visual solutions to common problems in design [70], analysis [67], or requirements [3, 157]. This direction seems to be very promising, as it offers higher level of reuse than just code. With this approach we can reuse whole software frameworks with several related classes or components (and possibly - with underlying code). What is very important, with the advent of the UML, the reusable pattern artifacts can be expressed in a uniform notation (see [150]) and can be reused by a vast number of developers trained in this standard modelling language.

Patterns offer reuse on an abstract level of generalised solutions. However, a very important aspect of reuse is the reuse of specific problems and their solutions. This means reuse of specific requirement, architectural or design models developed in past projects. Model reuse is a current research area (see eg. [201, 199, 194]). Model repositories offer an alternative to simple repositories of code components [162]. Software repositories based on UML offer standard notation and meta-model for repositories that can be searched with various methods based generally on comparing UML models (see eg.

[28] for one of the first attempts, and [188] for an overview of comprehensive reuse of models). UML offers standard mechanisms as stereotypes or tags that facilitate introducing reuse annotations, as discussed in [118]. This leads to a conclusion that UML is well applicable as a language for denoting artifacts in reuse repositories [141]. Specific models expressed in UML can be introduced into libraries and automatically reused (eg. sequence diagrams as described in [170]). Methods for reusing UML models include also reasoning [79] which will be described below.

Many approaches to searching reusable asset repositories exist. For instance, we can apply certain association rules and seek for these rules in the library [134]. Searching of the reuse repositories can be based on relevancy of assets compared with the initial queries [159]. This relevancy-based approach leads us to formulating queries by analogy [88] or using analogical reasoning [27]. An interesting approach to finding analogies between software artifacts is the use of the WordNet lexical database [27, 77] which helps in determining lexical similarities. By using analogy, past software artifacts can be sought for, by comparing them to initial rough sketches of the problem at hand. This gives us an advantage of implicit (as opposed to explicit) queries (see [228] and [8] for some additional insight). Such queries relieve the developers from additional work associated with formulating them. They can just create initial analytical or requirements models and seek for patterns in the library solving similar problems.

Software reuse is certainly very close to general reuse of knowledge. This means that Artificial Intelligence (AI) knowledge engineering techniques are well applicable here (see [75] for an example). Specific reasoning techniques applicable to software reuse include knowledge-based configuration (KBC), artificial neural networks (see [2]) and case-based reasoning (CBR). In KBC typically a combination of logical reasoning methods [15] and constraint processing technologies are applied for assembling technical systems from their parts. In KBC a descriptive and executable configuration model (ontology) is used for reasoning. Such a model specifies variability and dependencies between concepts. The configuration model is used by configuration tools for deriving configurations or products descriptions. The applicability of KBC techniques to software-intensive systems was recently examined (see [93]). Also other approaches like LaSSIE have been proposed, which uses the classifier of a description logic for querying [54]. It can be noted that such knowledge-based reuse means building an ontology that is equivalent to building a meta-model (see [55]) which brings this area close to UML reuse [34]. For instance, ontologies can be used to represent architectures [222] (another approach is shown in [229]).

Especially interesting reasoning technology in the context of this book is CBR (see [1] for a very good explanation of CBR). The idea of the ReDSeeDS framework is based on software cases. This is in accordance to CBR where cases describe specific problems and respective solutions, which are reused and revised within a knowledge management cycle. The main strength of CBR

lies in similarity-based retrieval techniques and in the combination with other machine-learning techniques. Considering this, CBR has been identified as a promising technology to implement software repositories and reusable asset libraries [208]. Several approaches have used case-based reasoning techniques in order to retrieve similar reusable software assets [66, 92, 180, 80, 78, 98]. For example, [140] demonstrate an automatic system for retrieval of hierarchically organized software assets (formed into software cases). Another example shows the use of CBR to facilitate reuse of object-oriented code [110].

Until the late 1990s the reuse of software mostly focused on what is called today opportunistic software reuse [17, 109]. Over the last few years work on software reuse strongly focused on software product lines (SPL) [221, 36, 76]. Software product lines strongly emphasise the need for engineering for software reuse - instead of just expecting to retrieve adequate components. This has led to very high rates of reuse: 95% (see [36]) in comparison to 40% (at most) with opportunistic reuse (see [164]). This success was however possible through limiting significantly the applicability space of software components. A middle ground is taken by domain-specific approaches. They aim at defining specific system development languages that cover the needs of functional area - rather independent of the individual software products. This approach is represented by so-called software factories' approach [84].

An important source of experience comes from the Software Engineering Laboratory (SEL) at NASA Goddard, which routinely achieves an astounding 75% or higher reuse level [44, 185]. The SEL have adopted a domain analysis approach, carefully studying and modelling to create reusable components. The process of domain analysis is similar to requirements analysis, where requirements are reviewed for past, present, and future space missions. Construction of applications for a specific mission then becomes more configuring components from a reusable asset library than new development. The key aspect of domain analysis is that requirements extend beyond a single project (see eg. [124] for another interesting approach). This prominent NASA example shows high effectiveness of requirements-driven approaches. There are quite few research approaches to this subject, and they include formal requirements specifications for reuse [161], requirements artifacts classification for reuse [48], analogical reasoning [204, 121] and requirements meta-model standardisation [116]. There also exist sparse attempts to reuse use cases and their scenarios [179, 219, 25, 227]. Finally, an interesting example of formal requirements reuse that allows for formulating software cases in engineering can be found in [224].

1.3.4 Software development methodologies and reuse

Modern software development methodologies went far away from the traditional waterfall approach. Most of them define a flexible, iterative approach which allows for proper reaction to common risks in a software development

project. Vast majority of them use appropriate modelling techniques and languages (mostly UML) to cope with software complexity. Despite these major similarities, contemporary general purpose methodologies are basically divided into two groups. Methodologies from the first group are usually called "formal", while those from the second group are called "agile". From the group of formal methodologies we can mention Rational Unified Process (RUP) from IBM [115], Microsoft Solutions Framework (MSF[2]), Application Lifecycle Management (ALM[3]) from Borland or independent Object-Oriented Process Environment and Notation (OPEN) [65]. Of numerous agile methodologies, several worth mentioning are eXtreme Programming (XP) [18], Feature Driven Development (FDD) [158], Agile ICONIX [175] and Crystal methods [41]. Formal methodologies offer a very comprehensive lifecycle process. However, being comprehensive they tend to become very hard to grasp by a typical developer team. Moreover, significant effort is needed to adapt such a methodology to the specific project. This is often done without necessary knowledge causing large unnecessary (formal) ballast to be produced. On the other side, agile methodologies offer a very lightweight process with a small number of core practices. This obviously relieves projects from the unnecessary ballast. Unfortunately, agile processes tend not to scale well to large projects. Improperly adapted they may reduce design activities causing the project to shift into "hacking mode", reducing development activities to coding. Moreover, agile processes do not tend to produce any documentation (like design models or requirements specifications) which is a serious problem when we want to apply reuse (see [142] for an attempt to change this).

The above methodologies are suitable for a whole spectrum of software development projects of variable size and problem domain. Unfortunately none of these methodologies offer a systematic process for reusing knowledge gathered in previous projects. It can be noted that formal methodologies tend to produce too much of this knowledge, and agile ones - too little. Regardless of this difference, general purpose methodologies give practically no guidelines on the issue of reuse in general and requirements-based reuse in particular. Due to this, dedicated reuse-oriented methodologies have emerged. First of these methodologies relied mostly on waterfall lifecycle (see [108]) and did not focus on reuse originating in requirements. In the last years, the reuse-oriented methodologies shifted their focus to software product line paradigm (see e.g. [124]), where reusable knowledge is gathered for a family of potential systems. This knowledge is usually based on domain analysis expressed through variable requirements. There exist several methodologies in this area. Most prominent of them include KobrA [14], FORM [107] and ConIPF [93]. These methodologies define software lifecycle in which reuse activities are inherently included into the activities of a software development project. Other attempts to define a software reuse method include generative programming

[2] http://msdn.microsoft.com/msf
[3] http://www.borland.com/us/solutions/

[26, 50] and knowledge management methodologies based on general purpose software development methods (eg. [195] which is based on RUP).

1.4 Going beyond the state of the art

As it was shown in the previous section, the idea presented in 1.2 has a sound technological basis in the current state of the art. However, in order to meet these objectives, important new research issues have to be resolved, thus advancing technology. It has to be stressed that the advancement in state of the art is needed not only through advancing research in the individual areas described above. The most significant achievement to be made is skillful combination of these areas resulting in a coherent framework supporting software industry in accomplishing its tasks more proficiently through reusing its experience. In order to describe prospective advancements in technology, we will start by describing advancements needed in individual areas. Then we will show how these advancements can be additionally enhanced by synergy of combined approaches.

After reading the previous sections, someone might argue that the main feature of the ReDSeeDS system is that it constitutes a "yet another" reusable asset library. It is true, but only to the extent that the ReDSeeDS Engine is in fact a kind of reusable software asset library. The main challenge for future research is to supply the software developers with a library which is easy to query and what is perhaps more important - that enables easy introduction of reusable assets. Unfortunately, present approaches to reusability require from software developers significant effort to prepare an asset to be reusable (see eg. [120]). This is the reason why very few software development organisations decide to invest in arranging a reuse-oriented lifecycle. Moreover, reusable assets are accessible through generalised and informal problem descriptions. This makes the reuse process tedious and subject to individual knowledge of skilled "librarians". Advancement of this project would be to offer the means for automating these tasks and making the whole reuse process "seamless" to the developers. The reusable assets in the ReDSeeDS framework would need no additional effort to be introduced in the library. This would be due to the fact that Software Cases stored in the ReDSeeDS repository would have exactly the same meta-model behind them as Software Cases produced during a software development project. Introduction of Cases would be as simple as copying all the models and code that form a case into the repository. No additional "generalisation" or "variability analysis" would be needed. "Software product lines" could thus be replaced with "requirements-driven reuse frameworks".

To advance technology in the area of reusable asset libraries we need to have better means than currently to seek for assets. In ReDSeeDS these means are based on requirements specifications. In order to be able to reuse software cases effectively (or rather: automatically), the requirements specifications

need to be defined very precisely. Only precision would allow for application of appropriate query mechanisms. This necessitates significant advancement in defining precise and comprehensive requirements specification languages. It can be strongly argued that such a unified language is an important step beyond the current state of the art. The value of this language would lie in a precise meta-model behind it allowing use of requirements for automatic querying and mapping into design. Comprehension of the language should be validated by ordinary people in real-life projects, which should give significant argumentation for using this language widely in the software development community.

Seeking for assets through requirements would have no value if we could not associate these requirements (problem statements) with appropriate solutions. Here, model-driven development comes into play and necessitates appropriate enhancements to the current technology. Currently existing modelling and transformation languages do not have the potential to define fully reusable Software Cases. These languages mostly concentrate on the design models of software and transformations between various levels of design. In ReDSeeDS, we need a modelling and transformation language that could also enable mapping from requirements into design. For this purpose, a combination of textual, graph-based and pattern-matching approaches is needed. Creating an efficient combination of these approaches is a challenge for future research. Another challenge in this area is to define a transformation language that is easy to use by software developers. Especially challenging would be to validate the language in practice for non-trivial applications.

An asset library being part of the ReDSeeDS framework needs efficient query mechanisms to facilitate reuse of stored cases. Non-standard mechanisms will most certainly be needed. Graph-based or textual queries will most probably not be enough to retrieve stored cases. This is due to high complexity of cases that comprise models interweaved with mapping and transformation relationships. An idea is thus to use also reasoning techniques known in the field of knowledge-based systems. Some good results in using Case Based Reasoning in software reuse have been shown. However, in order to implement the ReDSeeDS system we will certainly need a combination of reasoning techniques with graph querying. The assets subject to querying are mostly complex graphs and thus traditional CBR techniques would probably be not powerful enough to handle them. Thus, another challenge for future research is to combine reasoning techniques with graph querying to result in a powerful retrieval mechanism for complex model-based software cases.

Another problem that needs to be resolved when applying queries in the ReDSeeDS framework is the application of a marking mechanism. For the software cases to be easily reused, they have to be marked to show differences between the current problem and the retrieved problem. This is especially important, when the developers want to query the library with only partially prepared requirements models. Moreover, the markings need to propagate into design models and code to show clearly those places where changes are

necessary. The marking mechanisms are currently used in Software Product Lines to show variability of product families. These mechanisms can be used in ReDSeeDS, however the challenge here is to propagate markings along transformation paths as defined in the transformation language. This would necessitate a novel combination of techniques from software product lines and model-driven development. In addition, this combination would need to be built into the query mechanisms described above.

Having a complete reuse-oriented software development framework we certainly need to define a methodology to use it. As it was stated in the previous section, the currently existing general purpose model-based methodologies do not take into account such extensive reuse as offered by ReDSeeDS. This means that they need to be extended in the area of knowledge management. On the other hand, the reasoning-based methodologies are quite weak in handling models and their transformations. This leads to another challenge for research which is to combine and enhance existing methodologies to give guidance on using novel mechanisms introduced in ReDSeeDS.

To summarise, it can be argued that the prospective ReDSeeDS framework combines several areas of research never combined before in such a coherent way. General mechanisms of reusable asset libraries are combined with advances in requirements engineering, model transformation and querying. This combination gives a necessary synergy to create a system that facilitates efficient reuse of knowledge in software engineering. Advances in all the mentioned areas strengthen each other as it was shown in the preceding paragraphs. For example, advancing precise requirements models gives necessary means to advance Case Based Reasoning for software. Advancements in model transformations give the possibility of enhancement of query and reasoning technologies.

1.5 Potential impact on the software development industry

The impact of the proposed idea is directly associated with the usage of the ReDSeeDS framework in every-day software development and acquisition practice. It can be strongly argued that the organisations applying the prospective results of research described in this book would be able to improve their procedures in these areas. These changed practices would significantly enhance the capability of these organisations to reuse knowledge produced in previous software development projects. This pertains to both the organisations that produce software and that acquire it, and includes organisations (communities) having responsibility for creating open-source software.

The area where ReDSeeDS can have significant impact is software acquisition and maintenance on the public budget levels. For instance, the European governments and local authorities spend billions of Euro in this area, and these figures grow systematically as new e-government initiatives emerge. A

prominent example here is the British government budget for IT (which is mainly spent on outsourced software development and maintenance), as described in [12]. The budget is estimated at 2.6bn GBP, with high percent of it allocated to outsourced software development projects. Unfortunately the British National Audit Office (NAO) reports many of these projects being failures in terms of using public funds. On the other hand, NAO also finds and studies successful projects: "A number of our reports have drawn attention to projects that experienced problems. Not every government IT project, however, experiences difficulties; many are successful. To understand why these projects avoided the pitfalls that befell others, the study will examine successful IT-enabled business change projects in both the public and private sectors." In addition to this, we can also cite Ian Watmore, head of the British Government's eGovernment unit: "We are already doing a lot of good things. The question is how we can make that more ubiquitous." The impact of research directions presented in this book is exactly concentrated on making the knowledge about "good project" results ubiquitous, and this is not only on a single state, but also - on the international level. The ReDSeeDS framework is aimed at capturing results of such projects (Software Cases) and reusing them in similar environments.

ReDSeeDs gives very important new possibilities of optimising spendings for software acquisition described above. It includes means for improvement in software acquisition procedures. An important means to enhance capability maturity in this area is efficient management of requirements - not only on a level of single organisation (like a local self-government authority) but more globally on a level of problem domain areas (like e-government applications for self-government authorities). This means that a single organisation (be it governmental, public or private) would have means to reuse the knowledge about business processes and possible software requirements from other such organisations or from a central consultancy authority. This would be possible by applying publicly (openly) available ReDSeeDS frameworks for various problem domains. This has a significant potential for strengthening the international community by giving it means to reuse knowledge about public administration processes served by software systems, throughout different states. On the other hand it would not compromise the ability to maintain distinctions between various states, as these distinctions could be reflected through differences in user requirements for different states.

The above would be accomplishable by the requirements-driven characteristics of the ReDSeeDS framework. The organisation could formulate its initial requirements using the ReDSeeDS Language, and then a similar Case (or Cases) would be retrieved from a publicly available ReDSeeDS Engine. After that, tenders can be invited for the development of the required system. The bidding companies would offer to build a system which is a modification of a currently existing one. Moreover, the difference between the new and reused system would be clearly marked by the ReDSeeDS engine. This would

give good means to calculate the costs of preparing changes, which should be significantly smaller than with traditional approaches.

It has to be stressed that the potential impact would also relate to commercial IT and software development organisations. Such organisations could use ReDSeeDS to optimise their development efforts through reusing past cases thus shifting from a "workshop" to "factory" levels of producing software (see [190]. This factory would work by supplying it with an initial sketch of user requirements, which could be quickly matched with an existing case and also quickly re-developed to fulfill the new or modified requirements. This could give significant new possibilities for even small organisations to operate on the global market.

To summarise it can be stated that reusable software cases based on precise (yet understandable) requirements can reduce levels of effort by adding higher levels of automation and minimising rework caused by misunderstanding of user needs. If applied by public and private sector organisations, this approach could lead to significant enhancement in capabilities for producing and acquiring software in various domains. It thus means that either the software acquisition budgets could be reduced (eg. by billions of Euro in the European scale) or (rather) more software in various domains could be produced thus improving living conditions for people.

1.6 Structure of the book

This Chapter introduces the idea of a requirements-driven software reuse system. It shows that many problems faced by the software industry originate in lack of repeatable solutions and inability to utilise already formulated software solutions in the current project. We need certain, easy to use mechanisms in place that would allow for systematic reuse of software. By analysing the current state of the art we can see that many elements of these mechanisms are already available. We need to combine them to form a coherent whole. In order for this combination to become possible specific new research issues have to be solved. In the following Chapters we can find a detailed discussion on the issues that lead to constructing a comprehensive requirements-based reuse framework.

In Chapter 2 we can find a description of the framework, with mechanisms and tools that support it (and generally - make it possible). This Chapter starts by introducing a vision on how to organise the reuse process in the context of a software development organisation. It follows with a detailed description of the process that such organisations should follow in order to achieve significant return from reuse practices. This process is supported by a software case specification language that allows for building coherent reusable artifacts. Finally, a sketch of requirements and architecture for a tool supporting the process is given.

While Chapter 3 defines the reuse process formally, Chapter 2 gives a more practical approach to formulating reusable software cases. It introduces concrete syntax for individual elements of such cases: requirements, architecture and detailed design. This syntax is used to formulate software cases in a systematic way. Techniques for transforming models to form a coherent path from requirements to code are given. These techniques can be used regardless of whether the reuse process would be implemented or not.

Chapter 4 returns to more formal discussion on mechanisms allowing for requirements-based reuse. It gives a detailed description of three crucial elements of the reuse framework. Two of these elements elaborate on the metamodel of the software case language introduced formally in Chapter 2 and described for practical use in Chapter 3. The first of these important elements is a requirements specification language. The language presented in this Chapter allows for producing coherent, queryable and transformable requirements which is necessary for fulfilling the vision from Chapter 1. The second element is a method for transforming these requirements into design and then into code. Here, specific transformation rules are given and a simple transformation language is introduced. The third element described in Chapter 4 is the retrieval mechanism and specifically a query language. The query language presented in this book is suitable for formulating queries that allow for matching requirements models formulated in the requirements specification language mentioned above.

The final Chapter gives a summary and discussion. Possible directions for future research are also given. Moreover, some details on current activities leading to the construction of the described software reuse framework are presented.

2

Mechanisms for requirements-based model reuse

2.1 Vision for organising software reuse

In the previous chapter we have described the idea of a requirements-based model reuse system. We have briefly mentioned the mechanisms of recording and reusing software cases. We have also defined the system as a combination of a language, an engine and a methodology. Now we need to describe the system and its mechanisms in more detail.

The role of the ReDSeeDS system is to help software development organisations in reducing costs through active reuse of past software ceases. In order for the system to fulfill this goal, it should efficiently support these organisations in their everyday software development activities. We should try to optimise these activities assuming the existence of appropriate languages and tools. Only after this we should design these languages and tools. This means that before we define the ReDSeeDS language or the ReDSeeDS engine we need to determine the lifecycle process for using ReDSeeDS.

The ReDSeeDS system can be seen as a software reuse strategy supported by comprehensive software reuse tools and reuse-oriented software case language. It can be argued that the main problem with existing software reuse strategies so far is the size of investment that needs to be undertaken to enable the reuse process. Thus, while designing a new reuse strategy we need to consider several topics important from the point of view of software development teams.

- **Compatibility with the existing software development processes.** The new strategy should easily fit into the current practice of software organisations. It should add as little additional activities as possible. These additional activities should be treated as very light "plug-ins" to the basic methodology (e.g. a RUP reuse plug-in, an XP reuse plug-in or FDD reuse plug-in). For these organisations that do not have a systematic methodology in place, the system should offer one, allowing to tailor it to the size and domain of the organisation.

- **Additional effort needed when formulating software knowledge.** This knowledge is produced during every software development project. It contains various artifacts written in various notations. It seems crucial to use notations that are commonly used throughout the industry. In the areas where such universally accepted notations do not exist, it is necessary to introduce comprehensible extensions to the existing languages. Ideally, the process of formulating software knowledge should be seamless from the point of view of software developers. This means performing exactly the same set of activities and using the same languages and tools as without the reuse strategy in place.
- **Ease of capturing software knowledge for reuse.** Usually, making software reusable is treated as a laborious task. It is thus treated as an additional investment and applied only when the perspective of reusing current artifacts is very probable. However, even then we cannot be certain that the investment will get a return. Many reuse efforts gain profit only when the artifacts are reused several times - so high are the efforts associated with making artifacts reusable. In an ideal world, the effort associated with capturing software knowledge should be minimal. It could ideally mean literally copying the current project workspace to an appropriate library. The cost should be minimal enough to make saving current software cases in a reuse library - an obvious routine.
- **Ease in reusing past software knowledge.** Software developers hesitate to reuse past knowledge for various reasons. Basically, all these reasons lead to high effort levels associated with finding and tailoring past artifacts. This can be again treated as an investment which might not give satisfactory return. It is generally very difficult to find past software knowledge applicable to the current problem specification. With such high effort rates, software developers would expect to be able to reuse as much artifacts as possible. Unfortunately, most software strategies offer reuse of low-level components. Even if certain higher level artifacts (requirements specifications, architectural designs or such) are reused, they need to be tailored significantly and significant effort is needed to determine changes in associated code (if at all available for reuse). We should thus postulate that as for capturing software cases, the efforts associated with reusing them should be kept to the minimum. This means that querying a library of past cases should be based on the artifacts that would be developed anyway in a normal development practice. Moreover, query results should give precise pointers to these places that necessitate rework thus minimizing efforts associated with tailoring.

Now, let us imagine a software development organisation that considers introducing a reuse strategy into its practice. The first question is usually: does it make sense from the technical point of view? The second question is: how much will it cost? And finally: what profits will it give? Technically, introducing a reuse strategy means that the organisation has to change in

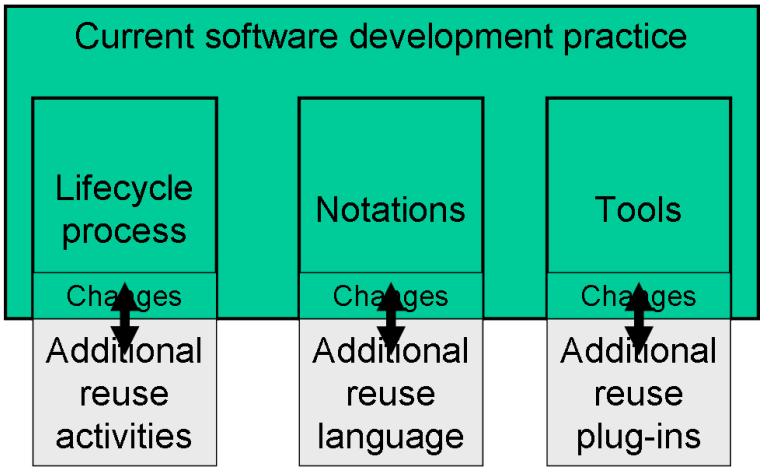

Fig. 2.1. Three technical areas for implementing a reuse strategy

three areas that constitute its core business. This is illustrated in Figure 2.1. The organisation's lifecycle process needs to be updated with additional reuse-oriented activities. Moreover, the organisation would usually need to change certain amount of its current practices in software analysis, design, testing and so on. The notations used by the organisation have to be updated with a language that allows for formulating, seeking and marking changes in software cases. It is also highly probable that the organisation would need to change some or even all of its notations used for producing software artifacts. Finally, the tools that the organisation uses would need significant enhancement. If enhancements to existing tools are not possible, the existing tools need to be changed.

It can be noted that applying reuse strategy in a software organisation faces two significant cost barriers. The first barrier is in investing in the three areas from Figure 2.1. The second barrier lies in applying reuse between projects. This is in contrast to implementing general software development methodologies. For these methodologies only one investment barrier exists. The organisations need to change their processes, notations and tools, but no additional efforts are needed to capture knowledge between projects. This difference is illustrated in Figure 2.2. The effects of implementing a general non-reuse oriented methodology can be seen in a significantly shorter period. For that reason, usually, even the promises for larger return on investment cannot convince such organisations to apply a reuse strategy.

What can we do to change this situation? The answer lies in the four topics described earlier in this section. We need a reuse system that is compatible with the state-of-the-art software development methodologies, and significantly reduces efforts in formulating, capturing and reusing software

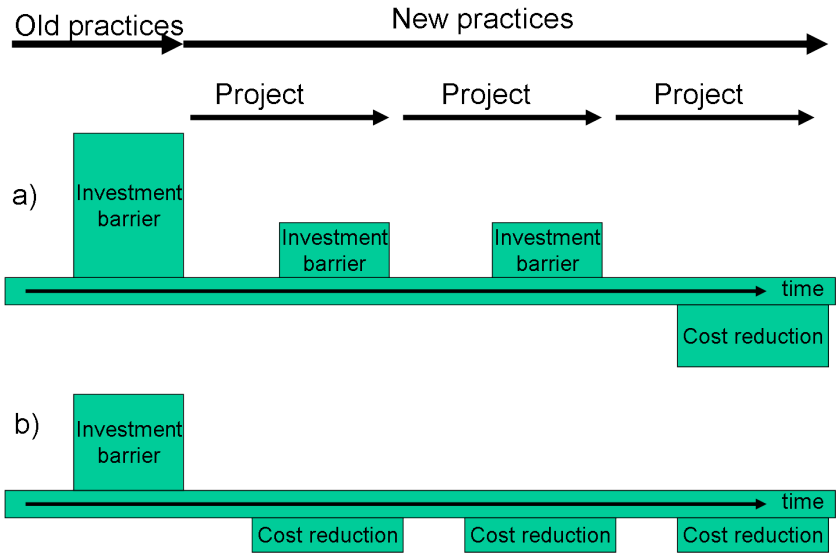

Fig. 2.2. Barriers in implementing a software development methodology: a) with a reuse strategy, b) without a reuse strategy

knowledge. This system should ensure that the initial investment in changing practices will be as small as possible. Moreover, in case the organisation invests in implementing a new general software development methodology, the investment in additionally setting a reuse strategy should be insignificant. Also the additional investment in formulating reusable software knowledge should be small enough to be practically dissolved in other activities.

In this book we want to prove that such a system is possible. In the following sections we will present the most important concepts associated with three combined elements of the ReDSeeDS system. We will start with the ReDSeeDS methodology and specifically with the methodology's lifecycle process. The methodology will give us the basis to formulate the requirements for the ReDSeeDS engine and design the tool's architecture. This architecture will consider the ReDSeeDS language described in more detail in the last section.

2.2 ReDSeeDS Methodology

In order to realise the vision presented in the previous section we need to construct appropriate tools that would enable "seamless reuse" practices. Only with such efficient tools we can make a dream of effortless reuse come true. However, before we start designing a software reuse engine we need to determine its functionality and language used to formulate reusable artifacts. These requirements should be derived directly from a lifecycle process that

would include software reuse activities. This lifecycle needs to be optimised taking into account the described vision.

2.2.1 Adding reuse activities to a modern software development lifecycle

As it was noted in section 2.1, the reuse activities should be as seamless for software developers as possible. In other words, these activities should "dissolve" in the software development process that they would normally use. This means that we should try do design a reuse lifecycle that is compatible with major modern software development methodologies. If we assume that an organisation uses one of these methodologies, the new "software reuse discipline" could become a natural extension (plug-in) to their current practices. If however, the organisation doesn't use any modern methodology, the implementation of a reuse lifecycle could be associated with introducing the new methodology. In any case it should be taken as a prerequisite that reuse activities have to be performed in a high software capability maturity culture (see. [184] or [214]), with precisely defined roles (see eg. [4]. This culture promotes repeatable process and repeatable structure of its artifacts which is an important condition for reuse.

Let's now analyse current software development methodologies. We will try to find their common elements in order to be able to design a common "methodological plug-in" for software reuse. This task seems to be quite hard as there is a large variety of methodologies available. Moreover, their proponents seem to fight a "methodological war" where the controversies are concentrated around the level of formality of a given method. From this point of view, we can divide software development methodologies into "formal" and "agile". Formal methodologies are usually designed and maintained by major companies or academic consortia. Agile methodologies originate from the famous "agile manifesto" (see: agilemanifesto.org). Various software development methodologies were listen in section 1.3.4.

Agile methodologies concentrate on face-to-face communication between people in a project. Formal methodologies on the other hand concentrate on communicating through deliverables (artifacts). For this reason, the first impression might be that formal methods are overwhelming in production of "unnecessary" artifacts. For agile methods we could have an impression that there is too little analysis and design leaving too much space for "code hacking". Methodologies use different notations for requirements management. Some of them use structured prose (MSF, ALM), other suggest using use cases (RUP, ICONIX), features (FDD) or user stories (XP). They also differ in defining specific activities, artifacts and tools necessary to develop a software system. Detailed analysis of differences between methodologies is certainly outside of scope of this book.

Despite the described differences, we can certainly identify a "methodological core" that is common to all of the contemporary methodologies. Most

of the methodologies use or support using UML as an analysis and design language. All of them use object-oriented paradigm for software construction and promote appropriate object-oriented techniques. Methodologies also have a common technical process. They use iterative lifecycle as opposed to waterfall which certainly facilitates reaction to change and reduces major risks. They promote requirements (or test) based project control. Finally most of the methodologies support component-based architectures, or at least recommend appropriate software modeling techniques that allow for managing complexity of systems. It can be also noted that some formal methodologies go in the direction of becoming agile (see eg. [114], [175] or [137]).

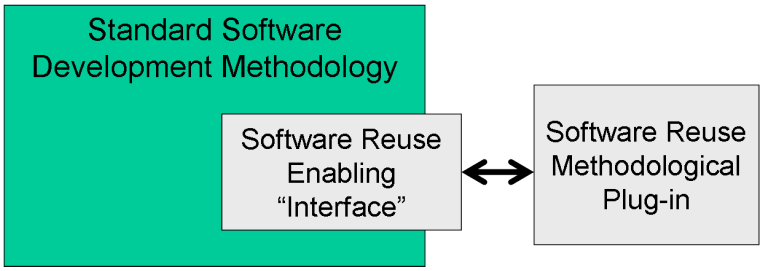

Fig. 2.3. Two core elements of a reuse-oriented software development methodology

Having identified important commonalities between methodologies we can try to identify the area where these methodologies can be extended with a reuse strategy. From this point of view, the base software development methodology has to enable maintaining "organisational memory". This means that the artifacts created during development have to be captured in electronic tools. Moreover, the structure of these artifacts has to be repeatable between projects in order to make their reuse feasible. It can be noted that capturing artifacts in a standard way does not yet constitute a reuse strategy. It can be treated as an enabler of such strategy, or a "software reuse enabling interface" to those activities that form the reuse process. This is illustrated in Figure 2.3. Thus, when describing our reuse-oriented lifecycle process we will clearly distinguish activities forming the reuse enabling interface and the reuse plug-in. We will also separately describe the core methodology.

2.2.2 ReDSeeDS Base Methodology

We will start defining our reuse-oriented lifecycle process by first sketching the standard software development methodology from Figure 2.3. We will not use any existing method but our base methodology will be quite close to Agile ICONIX [175] and FDD [158] which seem to offer a representative balance between formality and agility.

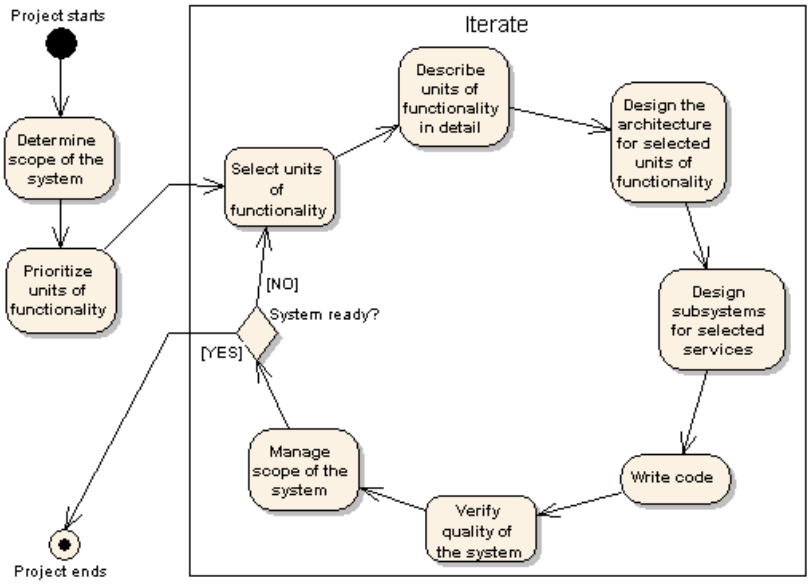

Fig. 2.4. Overview of the ReDSeeDS Base Methodology

Figure 2.4 shows a sketch of a typical iterative lifecycle process. We will call it ReDSeeDS Base Methodology (RBM). The process starts from determining the scope of the prospective system. With this activity, the resources for the project can be determined. The determined scope contains appropriate units of functionality to be implemented in consecutive iterations. These units need to be prioritised for their importance for the client and architectural complexity of implementation. The units chosen for a given iteration, control the rest of the development process. For this reason, the process can be described as requirements-driven. After starting a given iteration, units of functionality are described in detail. Having these details, the development team can update the architecture for the added functionality (architectural design). Then, the affected subsystems are designed in order to fulfill necessary services (this can be called detailed design). Based on detailed design artifacts, the programmers can now write code in a chosen programming language. The system extended with new units of functionality can now be tested and verified by the users (quality check). The results of quality control are the basis for probable changes in the system's scope. This can be also affected by possible change requests submitted during the iteration. After setting the new scope of the system, the next iteration begins.

Of course, the presented standard lifecycle is far from being complete. Certain important activities (including project management) have been omitted

as being outside of scope of this book. Moreover, only some of the presented general activities will be affected with a reuse strategy. We will thus now concentrate on these affected activities leaving other, shown in Figure 2.4 only for completeness of description.

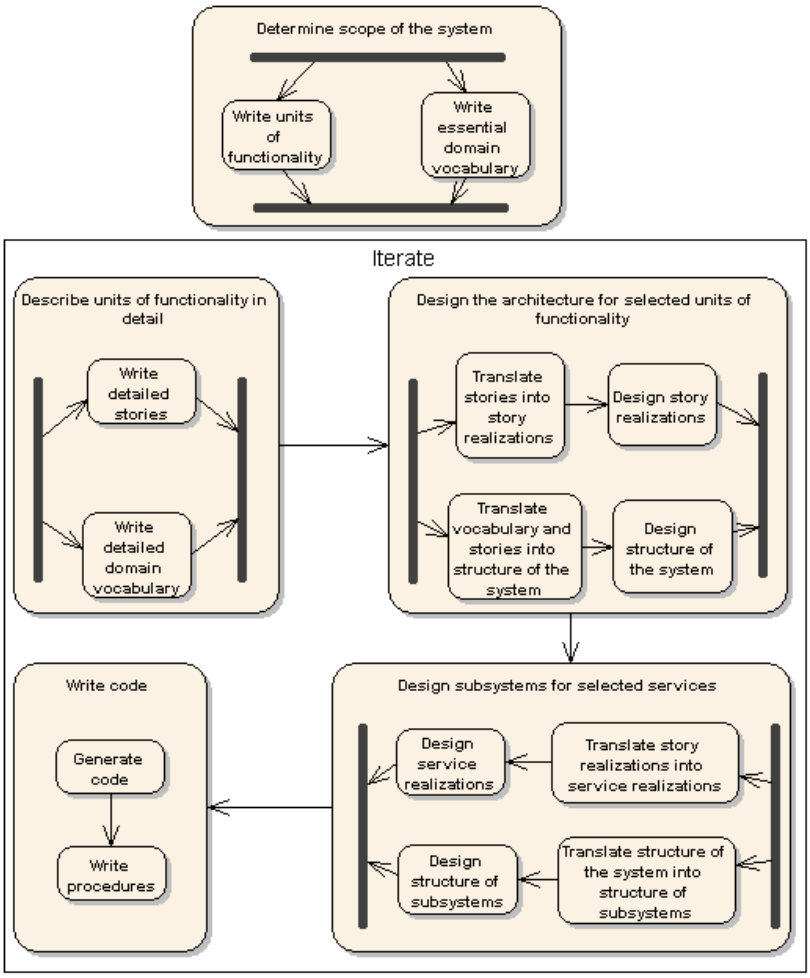

Fig. 2.5. Details of five activities producing knowledge about the software system

Figure 2.5[1] shows in "exploded" form, five activities from Figure 2.4. It can be noted that these activities produce knowledge about the developed software system. Moreover, these activities can potentially benefit from reusing

[1] In the figure we have omitted initial and final nodes for brevity.

past knowledge. Here we yet have a non-reuse oriented process, where appropriate intermediate artifacts are produced for the purpose of coping with the complexity of software and helping to build the final system.

Closer look at Figure 2.5 shows that generally the activities are divided into two parallel "paths". The first path constructs the dynamic aspect of the system, while the other one - the static (structural) aspect. These two orthogonal aspects, combined, form a complete software case[2]. Consecutive activities of the development lifecycle gradually build this software case. First, we build the description of the scope of the system. The scope consists of the dynamic scope containing units of functionality and static scope that encompasses essential domain vocabulary (see eg. [165, 62] for an insight). These two elements are built in parallel (see top activity in Figure 2.5) where notions defined in the domain vocabulary are used inside descriptions of the units of functionality. After the scope is known, we can start developing the system in an iterative cycle. Hence, we repeat appropriate activities building the system by units of functionality.

Every iteration starts with activities that describe chosen units of functionality in detail. This extends the developed software case with detailed stories and additional notions that form a detailed domain vocabulary. Again, these notions are used inside stories which describe the dynamics of the system in detail. After the functionality and vocabulary are known, the developers start designing the architecture for the described units of functionality. Stories are translated into story realisations (dynamics of the system). These story realisations are closely related with the structure of the system. The structure consists of components and interfaces used to describe the system dynamics. It can be noted that the developers first **translate** stories and vocabulary into story realisations and structure of the system. Only after performing this (possibly partially automatic) translation, they finalise the architecture by designing those elements that could not be translated directly from the requirements. A very similar process is performed for detailed design of subsystems. This time however, translation is done from the architectural level to subsystem level. In a given iteration, subsystems are designed to implement these services that participate in appropriate story realisations. By designing subsystems we finally reach the code level. Writing code (last activity on Figure 2.5) consists in automatically generating code from design (structure of subsystems and service realisations) and then writing (or just extending) procedure methods manually.

It is important to note the the presented base methodology (RBM) is concentrated on gradual building of a software system, where the produced artifacts are not only loosely coupled requirements specifications, design documentation and code. It concentrates on building a coherent piece of knowledge containing precisely specified models on each level of system's description.

[2] Actually, a complete software case contains also other aspects but we will omit them here for brevity and to show the general idea.

These models are translated between themselves and the knowledge about this translation is kept. Such a general software development lifecycle is certainly suitable to be extended with software reuse mechanisms.

2.2.3 ReDSeeDS Reuse Enabling Interface

Before we will describe reuse-oriented activities that extend the RBM we need to define important interface elements that need to be built into the base process. These interface elements should be built into normal software development activities. Exactly the same elements could then be used in appropriate reuse activities. By studying carefully the previous section, we can easily identify the elements of RBM that need to be adapted to enable reuse. These elements form complete software cases that result from consecutive activities in RBM shown on Figure 2.5. These are the pieces of knowledge that we can later reuse.

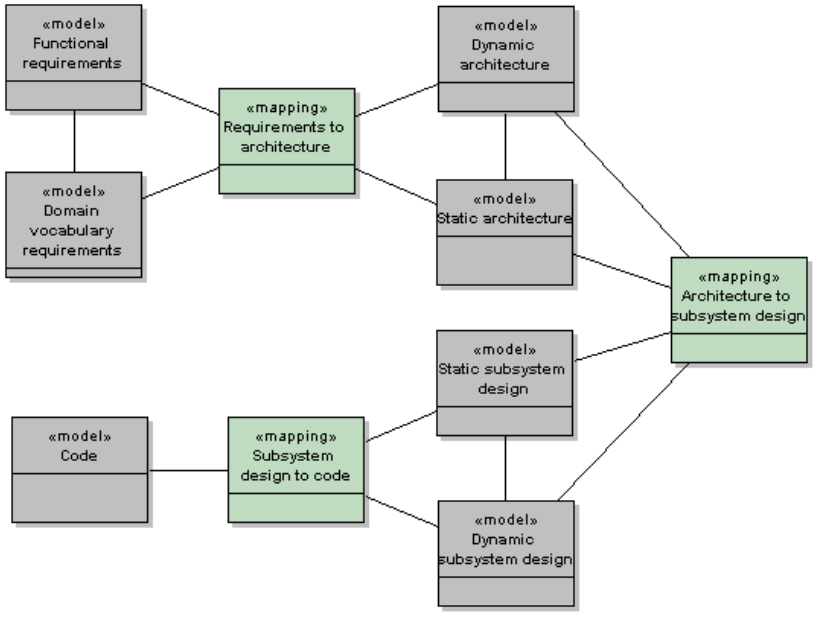

Fig. 2.6. Meta-model showing the structure of high level elements of a software case

Here we will define only the most important elements of what we will call ReDSeeDS Reuse Enabling Interface (RREI). This interface will describe the structure of software cases produced normally in RBM and that have features enabling their reuse. The main element of RREI is the mentioned case

specification language (see [169] for a new insight). It also consists of certain techniques that facilitate creating reusable cases. In this book we will concentrate on describing the language itself, leaving the description of techniques to more detailed work. The ReDSeeDS Language (RL) allows to describe software cases structured according to Figure 2.6. We will present the structure of the RL as a meta-model expressed in MOF [148]. It can be noted that the presented meta-model can be treated as an ontology for the software case knowledge.

In Figure 2.6 we can see the main pieces forming a software case, i.e. the elements that constitute technical knowledge collected during a software development project and possibly reused in future projects. It can be noted that metaclasses representing highest level elements of a software case are stereotyped as «model»s and «mapping»s. The first of these stereotypes distinguishes these pieces of knowledge that form complex abstractions of a certain domain or software system. These abstractions are formed of several interconnected elements (model elements) and allow for defining the system in a way understandable to the readers and possibly allowing for automatic processing. It can be noted that code is treated here also as a kind of model (written on the lowest level of abstraction). The second stereotype is attached to these software case elements that form descriptions of ways to translate (change) one «model» into another «model». This includes specifications to translate these models automatically or manually created links between model elements.

Models in Figure 2.6 are divided into four groups: requirements, architecture, subsystem design and code. The first three of these groups are still divided into two aspects: static and dynamic. Code has both of these aspects combined in a single model. Seven «model»s are connected through three «mapping»s. These mappings allow for translating models from the highest level of abstraction (requirements) to the lowest (code). With the mappings it is possible to form coherent software cases where every model has mapping (traceability) links to higher and lower level models. These links will enable requirements-based reuse (as described in the following sections) but they also improve reaction to changes in a software project. Every change on any given level of abstraction can be now easily propagated to other levels showing consequences of this change. The models and mappings from Figure 2.6 are described in more detail in Figures 2.7-2.10.

Figure 2.7 shows the most general elements of two requirements models. The functional requirements model is composed of many units of functionality. Each unit of functionality contains several stories. The domain vocabulary consists of notions that are used inside the definitions of units of functionality and stories. We can define these elements of the requirements model as follows.

- **Notion** - a word or group of words having specific meaning in the domain associated with the system to be built. Notions contain the notion name

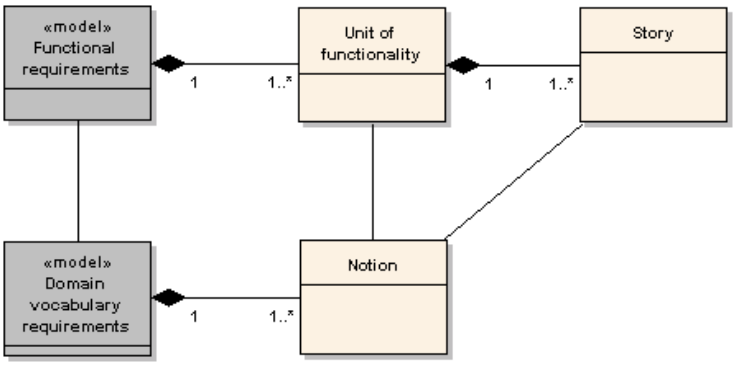

Fig. 2.7. Meta-model showing main elements of requirements

and notion description (definition). Notions can be used when defining stories and units of functionality.

- **Story** - a sequence of interactions between a specific type of user of the system and that system. The sequence starts when the user requests something from the system. The sequence is controlled by the system and leads to a single goal of significant value to the user. However, in a given story, this sequence might fail to reach the goal. The sequence of events in a story is defined with the use of notions.
- **Unit of functionality** - a group of stories starting with the same interaction of the user and leading to the same goal. In addition to combining several stories, units of functionality have an additional essential description common for all stories where the same notions as in stories (subset of notions) are used. Commonly used units of functionality are use cases.

It has to be noted that the above elements of the requirements model should facilitate formulating unambiguous and consistent specifications. Such specifications should be easily used to control production of the remaining models and at the same should serve as enablers of reuse. Thus, specific ReDSeeDS Language constructs for requirements form a precise meta-model with concrete syntax understandable by different participants of the development process. This meta-model and syntax are discussed in detail in Chapter 4.1.

The architectural models are presented in Figure 2.8. These two models describe the structure and dynamics of the system which should be consistent with the requirements. Thus, by analogy we divide architecture into static - derived mainly from the domain vocabulary, and dynamic - derived mainly from the units of functionality (functional requirements). It is worth noting that the architectural models described here show high level logical constructs of the system without going into details on how these constructs are built. Moreover, we do not describe the elements of physical architecture as being out of

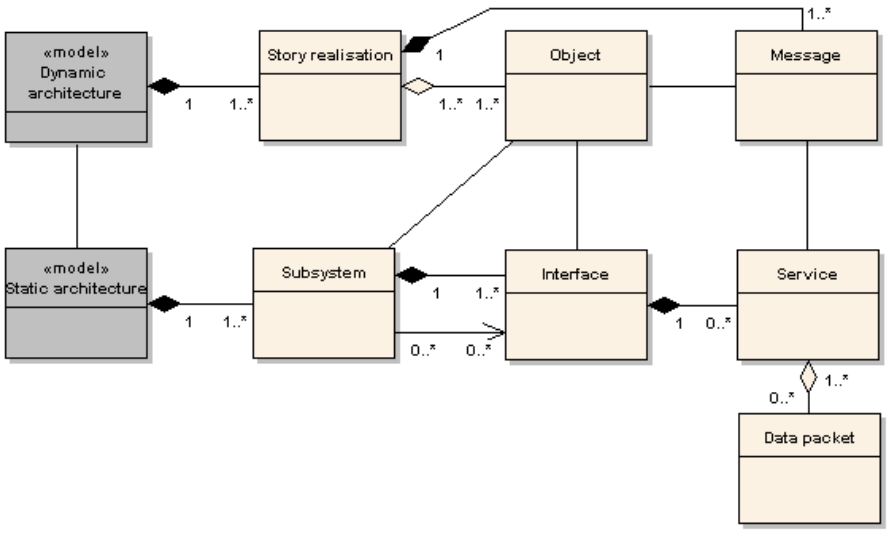

Fig. 2.8. Meta-model showing main elements of architecture

scope of this work. The static architecture is composed of subsystems. Subsystems contain interfaces and can use interfaces contained in other subsystems. Every interface may contain several services that realise data processing and exchange of data packets. The dynamic architecture contains generally several story realisations derived from units of functionality. Story realisations contain objects that dynamically exchange messages. Objects are instances of subsystems or interfaces. Messages are instances of services. More precise definitions of elements in Figure 2.8 is given below.

- **Data packet** - a container holding data exchanged between subsystems. The data contained in the packet should be consistent with the problem domain and thus derived from vocabulary notions.
- **Service** - a feature of an interface that defines a single unit of data processing to be performed by a subsystem (or subsystems). Services allow for exchanging data packets.
- **Interface** - a set of services grouped logically into a single model element.
- **Subsystem** - a logical unit of processing in the software system. Subsystems communicate with other subsystems through interfaces that they make available to other subsystems and interfaces that they use from other subsystems.
- **Message** - an event in a working software system consisting in passing control and data from one object to another object. Messages can evoke services which perform appropriate data processing.

- **Object** - a data processing element in a working software system. Objects can exchange messages.
- **Story realisation** - a sequence of messages exchanged between objects that realise a specific story or group of stories (most often: a unit of functionality).

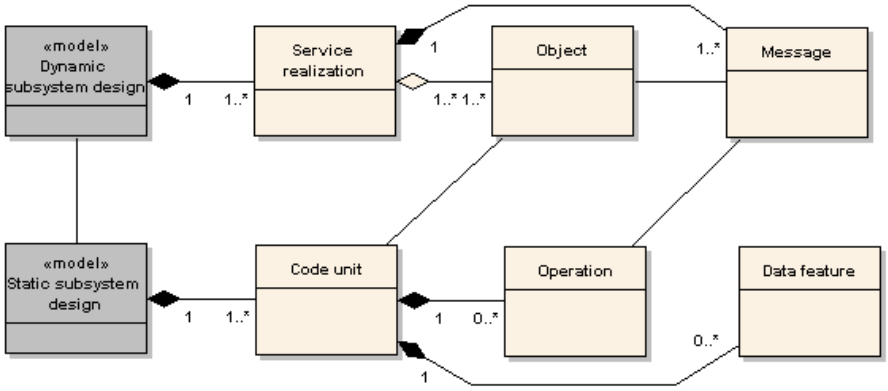

Fig. 2.9. Meta-model showing main elements of subsystem design

The high level subsystems from the architectural model need detailed design. Thus we introduce subsystem design models where every subsystem's structure and dynamics is specified. During a software development project, developers construct subsystem design models for every subsystem in the architectural model. The structure of these models is presented in Figure 2.9. It can be noted that subsystem design is very similar to architecture. The difference lies in scale and conformance with programming language constructs. Instead of interfaces and subsystems we have code units. These code units also serve as data holders (equivalent of data packets) consisting of data features. Code units contain operations similar to services on the architectural level. Due to these similarities we will not give detailed definitions of elements found in Figure 2.9.

To complete the definition of software cases we need to define mappings between models described above. The structure of these mappings is shown in Figure 2.10. This structure is common for all the «mapping»s shown in Figure 2.6. All of them are derived from a general model-to-model mapping (see appropriate inheritance relationships). Every mapping is composed of an automatic mapping definition and mapping instance. The automatic mapping contains a list of mapping rules. Each rule consists of a source template, target template and source to target translation. Mapping instance contains several mapping links which often are resulting from mapping rules (when they are

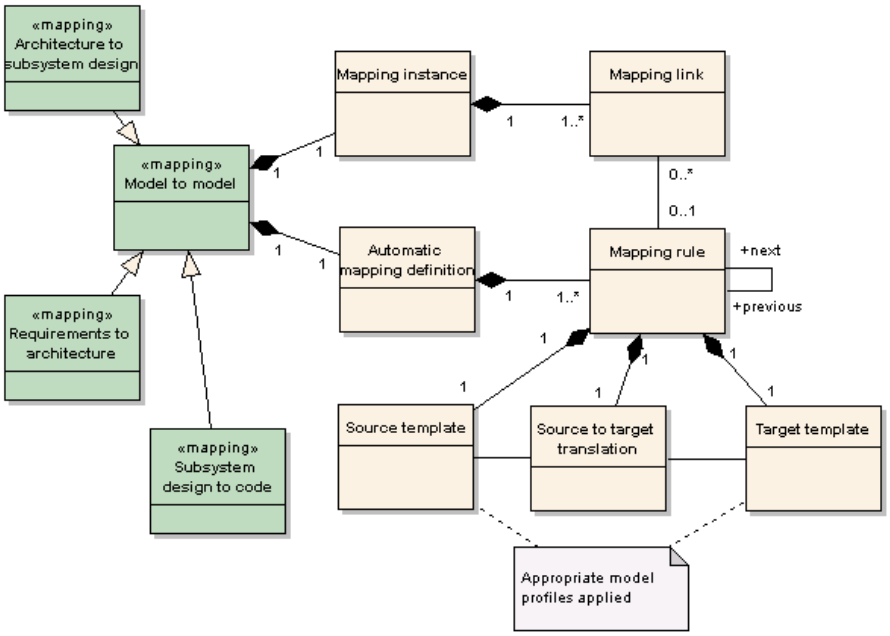

Fig. 2.10. Meta-model showing main elements of mappings between models

created by an automatic mapping). Mapping links are attached to elements from the source and target models (elements found in Figures 2.7-2.9). Brief definitions of the mapping elements are given below.

- **Source template** - a piece of abstract model showing configuration of elements to be found in the source model.
- **Target template** - a piece of abstract model showing configuration of elements to be constructed in the target model.
- **Source to target translation** - a set of instructions and constraints defining the way the source template should be mapped onto the target template.
- **Mapping rule** - a tuple containing source template, target template and source to target translation. This tuple is executed during automatic transformation process.
- **Automatic mapping definition** - a kind of "mapping program" consisting of a set of ordered mapping rules.
- **Mapping link** - a link between two elements from different models. This link is either generated through an automatic transformation process or determined by software developers.
- **Mapping instance** - a complete set of mapping links showing relationship between a source model and a target model.

The above meta-models for architectural and subsystem design and for model mappings lead us to the final code «model». The structure of these models is very important in order to assure good control over the complexity of the developed system. The architectural model should allow for good understanding of the overall system revealing only the most important elements. Detailed design model on the other hand reveals all the details of individual subsystems. On the code level we can see individual methods with appropriate algorithms implemented. While helping software developers to produce good quality software, the RL should also enable reuse of the constructed software cases. This necessitates the use of a widely used modelling language which could be consistent with the presented general meta-model. We need to note that it is important not only to use a proper general purpose modelling language but also profile appropriately the developed models (see [192]). This should prevent from abuse in using the language by developers (see [19, 20] for an excellent discussion). In this book we will use UML [154, 153] as our base language. We will use the concrete syntax of UML and its meta-model as the basis for constructing the presented «model»s. UML syntax will be also used to build a transformation language allowing to construct «mapping»s as defined in this section. UML with appropriate extensions defined in MOF will be used consistently in order to enable repeatability of software case structure. This in turn greatly facilitates reuse through appropriate reuse engines. The meta-model and syntax of the modelling and transformation language is presented in Chapter 3.

Finally, it can be noted that with the presented RREI (or rather: RL) we can use any existing modern software development lifecycle. Having such interface separates reuse activities from the rest of the process. This way, organisations that already have implemented a standard lifecycle can benefit also from the ReDSeeDS Methodology. At the same time they need not change their process dramatically. They only need to implement the "reuse interface". This also pertains these organisations that already implement or consider implementing one of the existing methodologies (like RUP, ALM, MSF, XP, FDD, and so on).

2.2.4 ReDSeeDS Reuse Methodology Plug-in

Having the above reuse enabling interface language we can define activities that shall allow for reuse of software cases specified in this language. These activities should be "plugged" into activities present in a normal software development lifecycle shown in Fig. 2.4. Basically, according to what we have discussed in section 1.2 (see Fig. 1.2) these activities are associated with «comparing similarity», «reusing» and «recording» of software cases. We need to determine during which activities in software development we should retrieve parts of stored software cases and store software cases we develop.

In Figure 2.11 we can see general reuse actions embedded inside activities of a general software lifecycle. These actions are performed together with non-

reuse oriented actions shown on Figure 2.5. Below we shall describe them in more detail.

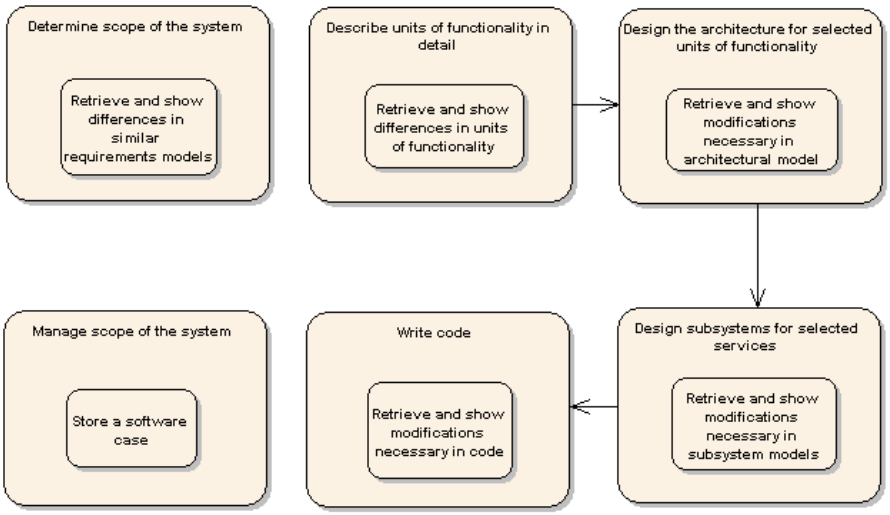

Fig. 2.11. Reuse oriented actions plugged into development lifecycle activities

- **Retrieve and show differences in similar requirements models** - query for software cases in a repository by finding similarities between cases based on similarity between their functional requirements (units of functionality) and domain vocabulary requirements (notions). After performing a query, the most relevant software case is retrieved. Differences between the current requirements model and the retrieved requirements model are shown (marked).
- **Retrieve and show differences in requirements elements** - retrieve detailed definitions of units of functionality and notions. After retrieving, the details are marked by pointing to elements which should be updated or are different in the current requirements model and the retrieved one.
- **Retrieve and show modifications necessary in architectural model** - retrieve architectural «models» associated through «mappings» with found similar requirements. The architectural models and the mappings are marked to show how they need to be modified in order to realise the current requirements in comparison to those retrieved from the software case library.
- **Retrieve and show modifications necessary in subsystem models** - retrieve detailed design models of subsystems found in the architectural model with mappings leading from appropriate requirements and architec-

tural elements. The detailed design models and the mappings are marked to show how they need to be modified in order to realise the architecture associated with the current requirements.

- **Retrieve and show modifications necessary in code** - retrieve code of subsystems developed for the stored software case. By analogy to the detailed design model, the retrieved code is marked to show how it needs to be modified.
- **Store a software case** - copy the current software case into a software case repository.

It has to be stressed that the above activities should be implemented so that minimal possible effort is needed to realise them, especially when we consider iterative and agile lifecycle process as mentioned in Section 2.2.1. This necessitates the use of an extensive automatic tool that implements appropriate model mapping and retrieval mechanisms (see [120]).

2.2.5 How should a software organisation change?

When we consider all the elements of the ReDSeeDS framework discussed above, we certainly come to a conclusion that software development organisation has to make an effort to adapt to it. This effort can be compared to efforts or investments illustrated in Figure 2.2. However, the major difference that the ReDSeeDS framework makes is that the investment should be quite similar to that without reuse strategy (Fig. 2.2.b) but promises returns of investment close or even greater than those with a standard reuse strategy (see Fig. 2.2.a) for every consecutive project.

Let's now summarise all the elements that a software development organisation should implement to change its practices in consistence with the ReDSeeDS Methodology. Here we can go back to Figure 2.1 which shows three technological areas where changes should occur.

In order to implement ReDSeeDS, a software producer should first apply a systematic iterative lifecycle (left column on Fig. 2.1). In many cases the effort (and investment) here is minimal as an organisation might already have such a lifecycle in place. This is the case for organisation which has already implemented a methodology like UP or XP. Implementing an iterative process is a prerequisite for any modern software development methodology. Thus, any organisation wanting to improve its processes would need to implement it anyway. In other words, the effort in this area is not associated with the ReDSeeDS Methodology. What ReDSeeDS adds is only the need to implement several additional activities as illustrated in Figure 2.11. When these activities are supported with a properly configured tool, they should not add any significant effort (see below).

Another element the organisation would improve when implementing ReDSeeDS is the practice of modelling when specifying requirements and designing

software systems. This practice consists in building such systems with a coherent software case specification language. This makes the process repeatable between different projects. A coherent language used to define various modeling artifacts significantly reduces efforts when configuring any new project (reduced learning curve etc.). Effort associated with this technological area (see centre column in Figure 2.1) is often already taken by the organisations. Many organisations already use a unified language to produce models (mostly UML). This means that applying ReDSeeDS would mean learning a modelling language plus certain additional elements (like a requirements specification language) and model well-formedness rules (eg. in the form of UML profiles). The effort of learning these additional elements can be minimised if the new notation would be based on the standard UML notation. Yet another issue associated with implementing the ReDSeeDS Language is the need to learn model transformation and mapping techniques. These techniques are already present in MDA (mentioned in Section 1.3.1) or MDD (Model-Driven Development). Applying ReDSeeDS would thus necessitate applying certain elements of MDD which is an effort that the software organisation should take into account.

Third area to consider when implementing ReDSeeDS is the area of CASE tools (see the right column of Fig. 2.1). This is coupled with the modelling language as discussed above. The use of a modelling language usually causes the need to use a modelling tool. In ReDSeeDS the use of such tools should become systematic and coherent. Thus, certain effort should be taken to teach software developers appropriate standard tools together with teaching the modelling language. New elements, specific for ReDSeeDS would include an engine allowing for storing and retrieving past cases. This engine should be tightly coupled with a modelling tool which can visualise the current and reused models. It has to be noted that such an engine, being complex by itself, should offer ease of retrieval and storage of software cases. This minimises the effort of performing reuse-oriented activities from Figure 2.11.

In summary we can state that in order to implement a true reuse-oriented software development process, an organisation should implement a systematic software development methodology. Within this methodology, the organisation should consistently use a standard modelling and transformation language. This standard language should contain certain elements enabling precise specification of requirements and generally should allow for capturing and retrieving software cases. The language with storage/retrieval mechanisms for software cases should be extensively supported by an appropriate engine (tool suite).

According to the current state-of-the-art, the use of methodology and language can be implemented through appropriate organisational efforts. Within this effort, appropriate existing CASE tools should be configured to enable capturing knowledge about models in a standard way. In order to enable reuse of software cases, the reuse oriented tool should add to this the capability of capturing knowledge about model mappings. Moreover, it should enable reuse

of captured cases with as little effort as possible. In the following section we shall discuss this in more detail.

2.3 ReDSeeDS Engine

2.3.1 Functional requirements for the ReDSeeDS Engine

Having defined a software development lifecycle and reuse-oriented language, we now need tools to support these two elements of a software reuse framework. As it was stated in the previous sections, these tools should form an "engine" to enable or facilitate activities associated with formulating and retrieving software cases. We shall now present functional requirements for this ReDSeeDS engine by using the UML use case model. In has to be stressed, though, that the requirements model presented below is far from being complete. Presenting all the detailed requirements is beyond the scope of this book.

Functional requirements (i.e. use cases) of the ReDSeeDS Engine can be divided into two packages. The first package is associated with formulating software cases, and the second with storing and retrieving them. The requirements for formulating software cases relate closely to the ReDSeeDS Language as briefly described in Section 2.2.3. On the other hand, storing and retrieving software cases is based on the methodology plug-in activities described in Section 2.2.4.

Formulating software cases

Formulating software cases is associated with following a coherent path from requirements to the resulting code. Here we shall describe three areas of this path: formulating the requirements, formulating the architecture and formulating the detailed design. In this path we shall identify certain services a tool should offer to software developers.

In the area of requirements specification we need support for formulating units of functionality, stories and notions as presented in Figure 2.7. Appropriate use case model is presented in Figure 2.12. Here we can see four general use cases described below. Terms used in these use case descriptions can be found in Figure 2.7 and in Chapter 4.1. Moreover, these use cases can be traced back to activities performed by Requirements Specifiers during the software development process. These activities were presented in Figure 2.5 (“Determine scope of the system” and “Describe units of functionality in detail”).

- **Formulate units of functionality.** This use case consists in creating a model that shows individual units of functionality with relationships between them. Units of functionality can be created, modified and deleted from the model by the Requirements Specifier. Relationships between units

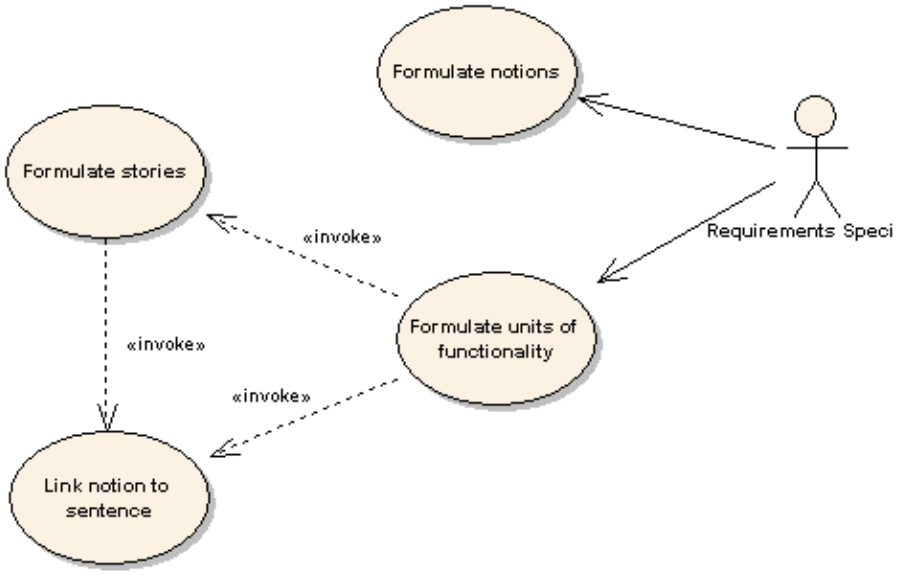

Fig. 2.12. Use cases for formulating the requirements

of functionality can be also depicted. When writing the name of the unit of functionality or its short description, the Requirements Specifier can ask for linking notions contained therein with notions from the domain vocabulary («invoke» *Link notion to sentence*). While formulating units of functionality, one can also «invoke» *Formulating stories* for chosen units of functionality.

- **Formulate stories.** In this use case, the Requirements Specifier can write stories in one of the notations handled by the Engine. Stories are written for a chosen unit of functionality. While writing stories, Requirement Specifier links phrases contained in sentences with notions from the domain vocabulary («invoke» *Link notion to sentence*).
- **Formulate notions.** Requirements Specifier can add and modify notions in the domain vocabulary. This use case allows also to attach appropriate phrases (with verbs or adjectives) to notions. Notions can be linked through appropriate notion associations.
- **Link notion to sentence.** This use case allows for linking notions to elements of sentences. Such links (or: hyperlinks) can be inserted into the sentence which is currently created or modified within the *Formulate stories* use case. To link a notion, the Requirements Specifier chooses a notion from the vocabulary. The system supports the Specifier by allowing to search for synonyms or words with similar meaning.

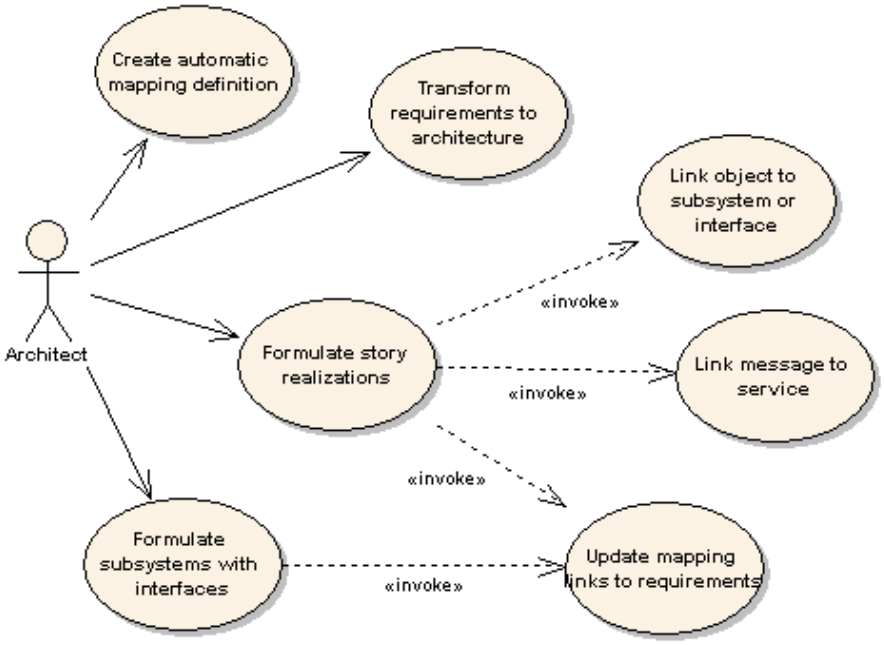

Fig. 2.13. Use cases for formulating the architecture

The Engine should also support the Architects in "Designing architecture for selected units of functionality" (see Fig. 2.5). Appropriate activities can be performed according to the use case model shown in Figure 2.13. The use cases contained there use terms found in Figures 2.8, 2.10 and Chapter 3.

- **Create automatic mapping definition.** With this use case, the Architect can create an automatic mapping definition with all the necessary mapping rules. This can be done both for requirements to architecture and architecture to design mappings. The system allows for creating source templates, target templates and source to target translations.
- **Transform requirements to architecture.** The Architect can start this use case when an automatic mapping definition is ready and available. He/she can also choose from previously prepared definitions. When a definition is chosen, the current requirements model (units of functionality and notions) are transformed into story realisations and subsystems with interfaces. This transformation results in an architectural model that is available for further "hand" modifications by the Architect. The resulting model should be appropriately mapped to the requirements. This mapping is an instance of the automatic transformation with appropriate mapping links.

- **Formulate story realisations.** The Architect can choose an existing story realisation or create a new one for an existing unit of functionality. With the chosen story realisation, the Architect can add objects and messages that show the dynamics of the system realising a given story or set of stories. While adding objects and messages, the Architect may *Link object to subsystem or interface* by «invoke»ing an appropriate use case. Same can be done for *Linking message to service*. When a story realisation is being formulated, there can be also *Updated mapping links to requirements*.
- **Formulate subsystems with interfaces.** This use case consists in modifying or creating subsystems with appropriately related interfaces. Relationships between these elements can be changed. Data packets transmitted through interfaces can be changed or created. These data packets can be added to the services of interfaces. Services can be added or their signatures modified. When subsystems and interfaces are being modified, there can be also *Updated mapping links to requirements*.
- **Link object to subsystem or interface.** For a given story realisation, the Architect may choose an object and associate it to an appropriate subsystem or interface. This subsystem or interface will serve as a classifier for that object. This results in making services of a given subsystem or interface available for associating with messages.
- **Link message to service.** For a given message, the Architect may choose one of available services of the target object classifier. This service will be then associated with the given message.
- **Update mapping links to requirements.** When changing any of the architectural models, the Architect can manually update mapping links that lead from appropriate requirements elements to architectural elements. This might be done for existing mapping links (e.g. created during automatic transformation). Additional links can be also added.

By analogy to supporting the Architect, the Engine should also support the Designer in "Designing subsystems for selected services" (see Fig. 2.5. Appropriate use cases are shown in Figure 2.14. Terminology for these use cases can be found in Figures Figures 2.9, 2.10 and Chapter 3. It can be noted that there is no specific use case for creating an automatic mapping definition between architecture and design. This was already done by the Architect.

- **Transform architecture to detailed design.** The Designer can start this use case when an automatic mapping definition is ready and available. He/she can also choose from previously prepared definitions. When a definition is chosen, the current architectural model (story realisations, subsystems with interfaces) is transformed into service realisations and code units. It is also possible that just some parts of the architectural model are chosen for transformation. This transformation results in a subsystem design model that is available for further "hand" modifications by the Designer. The resulting model should be appropriately mapped to the

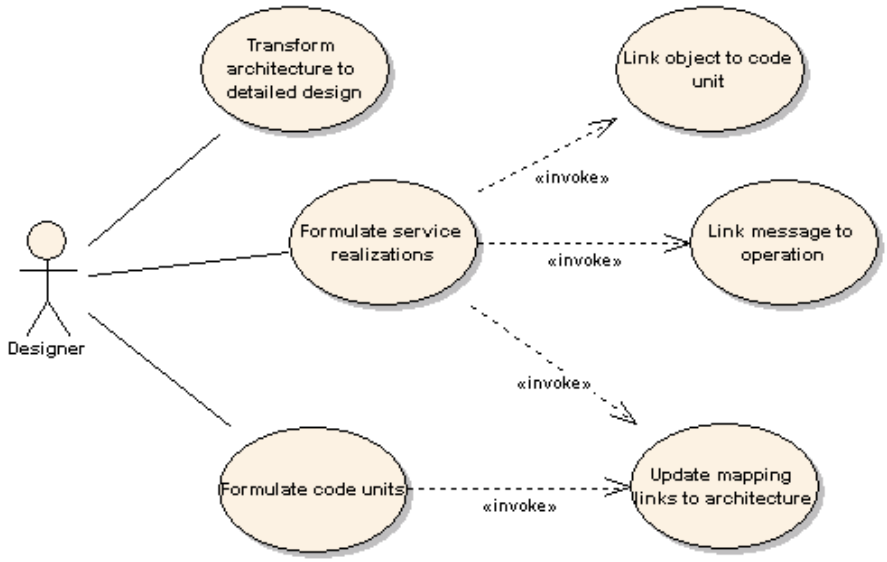

Fig. 2.14. Use cases for formulating the detailed design

architecture. This mapping is an instance of the automatic transformation with appropriate mapping links.

- **Formulate service realisations.** The Designer can choose an existing service realisation or create a new one for an existing story realisation. With the chosen service realisation, the Designer can add objects and messages that show the dynamics of the subsystem realising a given service. While adding objects and messages, the Architect may *Link object to code unit* by «invoke»ing an appropriate use case. Same can be done for *Linking message to operation*. When a service realisation is being formulated, there can be also *Updated mapping links to architecture*.
- **Formulate code units.** This use case consists in modifying or creating code units with their relationships. Data features and operations can be added to code units or modified. When code units are being modified, there can be also *Updated mapping links to architecture*.
- **Link object to code unit.** For a given service realisation, the Designer may choose an object and associate it to an appropriate code unit. This code unit will serve as a classifier for that object. This results in making services of a given code unit available for associating with messages.
- **Link message to operation.** For a given message, the Designer may choose one of available operations of the target object's code unit. This operation will be then associated with the given message.

- **Update mapping links to architecture.** When changing any of the subsystem design models, the Designer can manually update mapping links that lead from appropriate architectural elements to subsystem design elements. This might be done for existing mapping links (e.g. created during automatic transformation). Additional links can be also added.

It can be noted that the above use cases do not allow for generating code out of design models. Such functionality is available in existing tools.

Storing and retrieving software cases

The Engine supports various developer roles in storing and then retrieving software cases. With this support, significant parts of stored software cases can be merged into the current software case. This merging is based on similarity markings done by the Engine. The marking in turn, can be performed thanks to mapping links that exist between parts of the stored software cases.

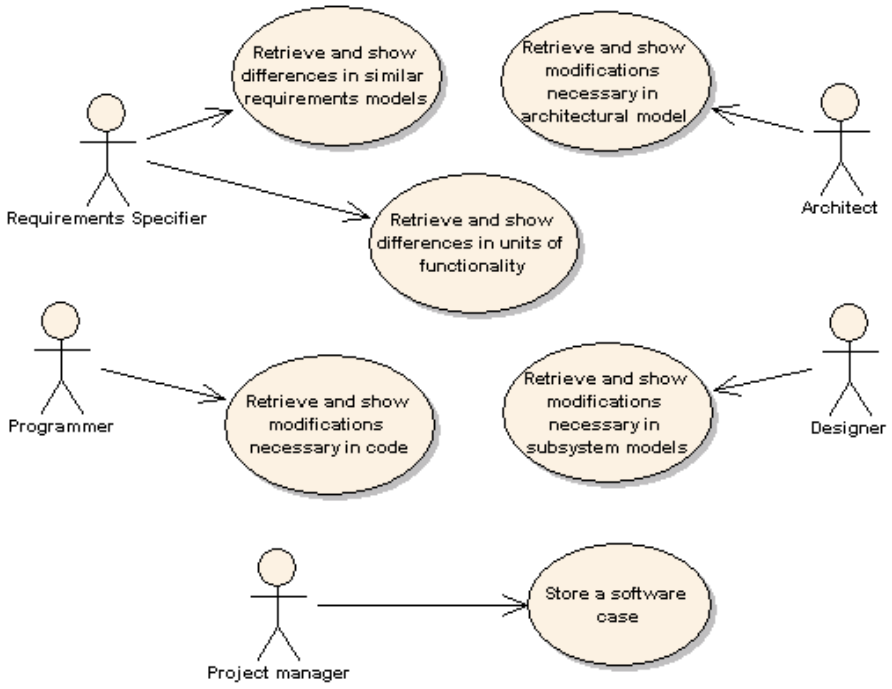

Fig. 2.15. Use cases for storing and retrieving software cases

- **Retrieve and show differences in similar requirements models.** This use case can be started when at least a partial requirements specification for a new software case is ready. The Requirements Specifier asks the engine to search for similar stored software cases. A list of software cases with similarity measures is shown. Then, the Requirements Specifier examines the software cases. If satisfied with one of the software cases, Requirements Specifier chooses it and the Engine retrieves the requirements part of the software case into the current workspace. After this, the Requirements Engineer merges the current requirements model with the retrieved one. This is done by hand in a model editor.
- **Retrieve and show differences in units of functionality.** This use case can be started when a complete requirements specification for some units of functionality (eg. for one of the iterations) is ready. Then, the Requirements Specifier can ask the engine to search for similar units of functionality. The Engine then shows a list of current units of functionality with attached lists of most similar stored units of functionality with similarity measures. Then, the Requirements Specifier examines the similar units of functionality, and chooses the most relevant. After this, the Engine clearly marks differences between most relevant stored units of functionality and the current units of functionality. The Requirements Engineer can still modify the current units of functionality to make them closer to the stored units of functionality in order to allow for greater levels of reuse.
- **Retrieve and show modifications necessary in architectural model.** This use case can start when a complete requirements specification for a significant part of the system functionality is ready. The Architect asks the Engine to show relevant stored software cases in their Architectural parts. The Engine shows relevant software cases with similarity measures. The Architect can examine the stored architectural models. These models are marked by the Engine through pointing out story realisations, objects, messages, subsystems, interfaces, services, data packets (in general: elements of the architecture) that would need rework. This marking is based on tracing from the stored requirements model to the stored architectural model. After choosing one of the models by the Architect, the Engine transfers it, together with the markings, to the current workspace.
- **Retrieve and show modifications necessary in subsystem models.** This use case can be started when a significant part of the architectural model is ready. The Designer asks the Engine to show relevant stored software cases in their Subsystem design parts. The engine shows markings pointing out those service realisations, objects, messages, code units, operations, data features that would need rework. This marking is based on tracing from the stored architectural model to the stored design model. After choosing one of the models by the Designer, the Engine transfers it, together with the markings, to the current workspace.
- **Retrieve and show modifications necessary in code.** This use case can be started when a significant part of subsystem desing for a particular

subsystem is ready. The Programmer asks the Engine to show relevant stored software cases in their Code parts. The engine shows markings pointing out those elements of Code that would need rework. This marking is based on tracing from the subsystem design model to code.

- **Store a software case.** This use case can be started then a software case is judged as complete, and all the necessary tests have been successfully passed. The Project manager asks the Engine to store the current software case. The Engine stores the software case by copying its complete contents into the software case repository.

It has to be stressed that the above retrieval use cases can be interleaved with use cases from the "Formulating use cases" package. After retrieving certain elements of a stored software case, the developers should perform activities associated with merging the query results with the current working model. This merger should be done using standard use cases to formulate a software case in various areas.

2.3.2 Architecture of the ReDSeeDS Engine

In order to fulfill the functionality presented above, the Engine has to be composed of several components. These components are presented in the following section. The components cooperate in order to realise use cases and these realisations are presented in the last part of this section.

Structure of the Engine

The ReDSeeDS Engine is designed using a tiered (layered) approach. We shall present the structure of the Engine by using a four tier framework:

- Presentation Layer – This layer presents results and accepts data from the users. Components on this layer contain code units that handle individual screens. These code units can generate the screens and accept button presses. The presentation layer should be responsible only for presenting and accepting data from the user.
- Application Logic – This layer controls the whole application by asking other layers for actions in a specified sequence – according to the functional requirements. Components in this layer contain code units that manage appropriate sequences of events defined in use cases. These code units know what to do when a certain button is pressed in a certain situation. They also react appropriately depending on the state of the system (handled by the business logic). In summary, Application Logic instructs "everyone" what to do and when.
- Business Logic – This layer performs data processing actions as asked by the application logic. Actions performed by the business logic are consistent with the business rules defined in the vocabulary requirements. Components in this layer contain code units that represent business vocabulary

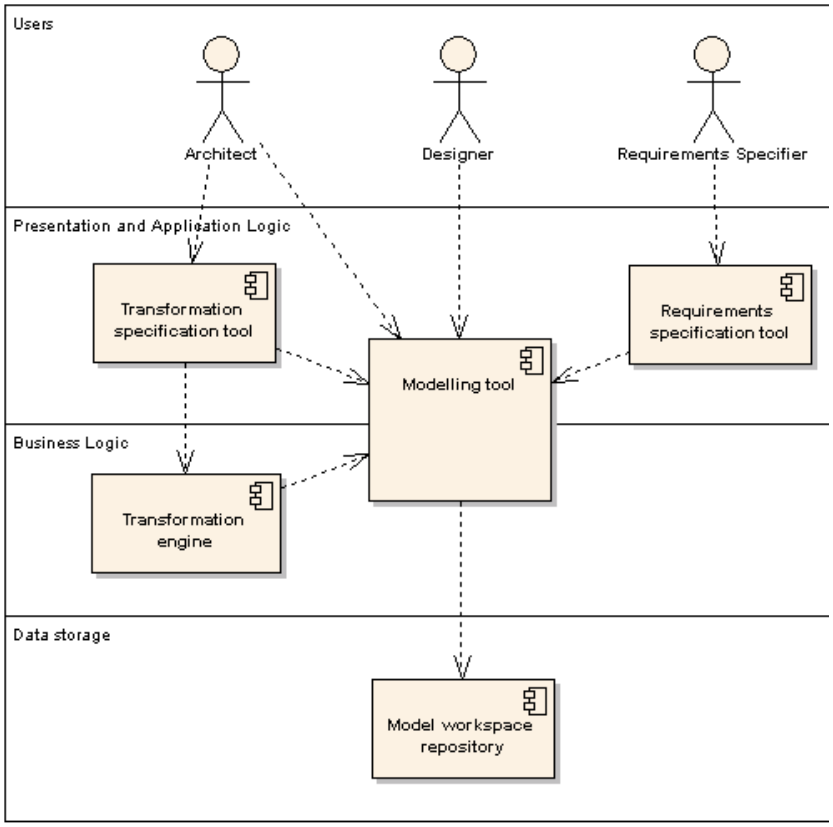

Fig. 2.16. Component diagram showing structure of the Engine for software case specification

notions. These code units have operations that process data contained in their objects. Business logic is the right place to integrate applications. Different applications should integrate by asking for actions on this layer.

- Data Storage Layer – This layer allows for persistent storage and retrieval of data whenever it is needed by the business logic. Components in this layer contain databases, file stores, data repositories, knowledge bases etc. Relational database components (the most common) contain tables with relationships. Such components can handle queries in a query language (eg. SQL).

In the proposed architecture of the ReDSeeDS Engine, the Presentation Layer and Application Logic Layer are closely related. Thus, we will show these two layers as a single layer.

Figure 2.16 shows components that take part in specifying software cases. The top layer of the diagram shows all the involved groups of users. Note that for brevity we have omitted the Programmer (as using the Engine in the same way as Designer).

The Requirements Specifier uses the Engine through a `Requirements specification tool`. This tool uses the functionality of the `Modelling tool`. The modelling tool is basically a standard UML modelling tool, chosen among available on the market. An important feature of this tool should be the existence of a comprehensive Application Programmer's Interface that allows for integrating with external applications or plug-ins. This modelling tool connects to a `Model workspace repository`. Normally, this repository is part of the modelling tool setup. However, we have emphasised this component as it includes the meta-model facilities offered by the chosen modelling tool. The `Modelling tool` is equipped with an extensive interface that allows for storing and retrieving models defined in other tools (like the `Requirements specification tool`). This interface handles models as defined throughout this book with appropriate meta-models.

The `Modelling tool` is used extensively by the Architect and by the Designer. They use it as a normal modelling tool but with restrictions set on the meta-model (eg. by using a UML Profile) as specified in this book. In addition to this tool, the Architect uses a `Transformation specification tool`. This tool offers functionality to define and run automatic mapping specifications. It is generally equivalent to the `Modelling tool` in the sense that it also allows for handling models but these models pertain transformation definition in the language as sketched in Figure 2.10. The tool uses a specialised engine (`Transformation engine`) that performs all necessary transformation operations on the models stored in the `Model workspace repository` through the `Modelling tool`.

Figure 2.17 presents these components of the Engine that are responsible for storing and retrieving software cases. The `Storage and retrieval tool` is used by all the participants of the development process. The developers (Architects, Designers, Requirements Specifiers, Programmers) use it to retrieve relevant parts of software cases similar to the current requirements model. The Project managers use it to store the current software case whenever a decision is made that it is complete and of good quality. The tool uses an extensive `Software case storage and search engine`. This engine has specific algorithms implemented that allow for retrieving software cases. These algorithms should be based on advanced graph comparison and case based reasoning techniques (and in general – AI techniques). The engine operates on the current software case on software cases stored in a repository. The current software case is handled by the `Model workspace repository`. The repository is stored in a `Software case repository` component. This last component has certain functionality to handle a specialised query language, oriented on querying graphs.

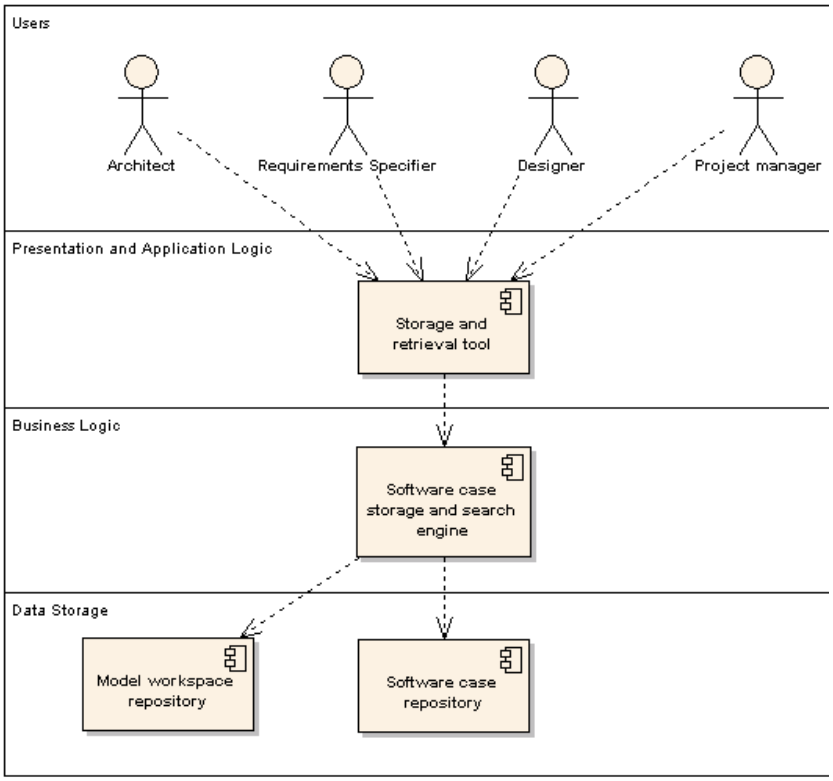

Fig. 2.17. Component diagram showing structure of the Engine for software case storage and retrieval

Dynamics of the Engine

The purpose of specific components of the ReDSeeDS Engine can be better explained by illustrating their usage in use case scenarios. Here we shall illustrate some of the scenarios with UML sequence diagrams that show interactions between components. The chosen scenarios are most representative for various functionalities specified in section 2.3.1.

When describing the realisations of use cases we shall not show all the details of user interaction with the system. This would introduce a lot of detail and hide the fundamental communication paths between components. Thus, only most important user actions will be shown here. More detailed sequence diagrams are left to be designed by the Engine developers.

Formulate stories. (see diagram in Figure 2.18)
In order to formulate a story, the Requirements specifier should interact with a `Requirements specification tool` ('Start formulating stories'). The user can ask to introduce a new story, giving its details (this interaction between

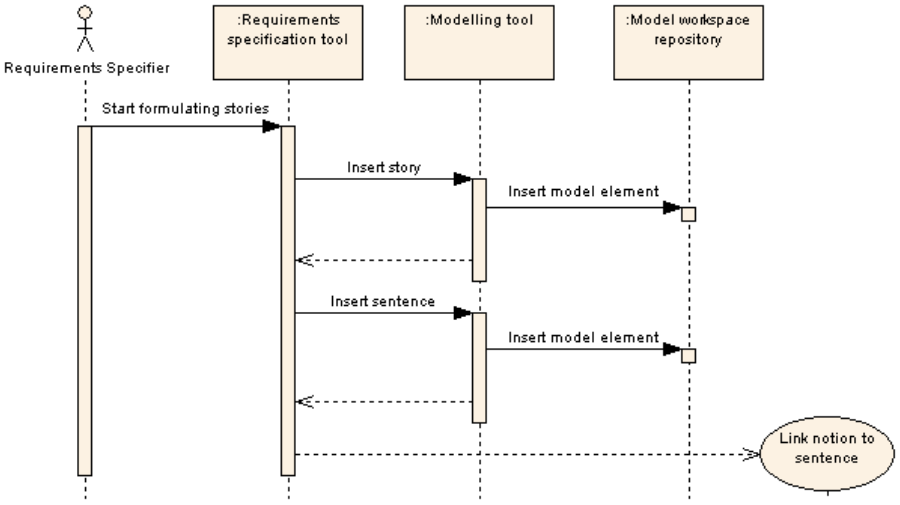

Fig. 2.18. Sequence diagram showing Engine dynamics for the "Formulate stories" use case

the user and the tool is not shown). After the details of the story header are introduced, the `Requirements specification tool` communicates with the `Modelling tool` to 'Insert story'. This is in turn performed by inserting an appropriate model element into the `Model workspace repository` ('Insert model element'). It can be noted that a story has to be mapped appropriately by the `Modelling tool`. This mapping is done from the software case meta-model for requirements (see Fig. 2.7) to the standard meta-model of the modelling tool (normally – the UML meta-model).

When the user wants to insert a new story sentence, the situation is analogous. After some dialogue with the user, the sentence is formed within the `Requirements specification tool`. Then, this sentence is inserted into the `Model workspace repository` through the interface contained in the `Modelling tool`. It can be noted that while formulating sentences, another use case can be invoked («invoke» Link notion to sentence), that allows for adding appropriate links within the created story sentence. This is shown on the diagram as a message to start this other use case, which is presented below (note that this notation is an extension, not present in standard UML).

Link notion to sentence. (see diagram in Figure 2.19)
When the Requirements Specifier wants to add a new link ('Start linking notion to sentence'), the `Requirements specification tool` asks for the notions to be linked ('Get notions'). The `Modelling tool` gathers all the relevant notions contained in the `Model workspace repository` ('Get model elements'). It should be noted that the 'Get notions' operation should act context sensitive. This means that the notions are gathered based on the

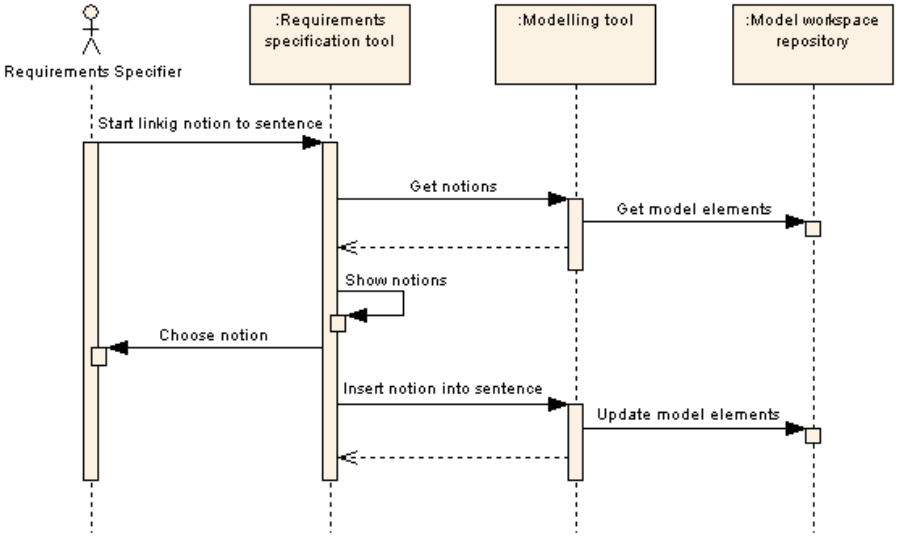

Fig. 2.19. Sequence diagram showing Engine dynamics for the "Link notion to sentence" use case

sentence element type and certain initial data given by the user (like the starting characters). This interaction between the two tools is performed while the user types sentence text.

After the notions are gathered, he `Requirements specification tool` shows them to the user ('Show notions) who can choose one of the notions ('Choose notion'). After this, the link is inserted into the current sentence and this information is stored into the `Model workspace repository` ('Insert notion into sentence' and 'Update model elements').

Create automatic mapping definition. (see diagram in Figure 2.20) This use case encompasses all the functionality associated with creating the mapping definition. Generally, every new mapping definition element is created in a specific dialogue with the user (Architect) which is not shown on the diagram. What is shown is that whenever a new element is created, it is inserted into the `Model workspace repository` through the `Modelling tool`. It can be noted that by analogy to specifying the requirements, the `Modelling tool` has to map the mapping definition meta-model (see Fig. 2.10) into its internal meta-model (usually the UML meta-model).

Transform requirements to architecture. (see diagram in Figure 2.21) When the Architect wants to perform a transformation ('Start transformation'), the engine allows to pick the appropriate transformation model and the suorce and target model packages (this dialogue is not shown in the diagram). Then, the `Transformation engine` gathers all the necessary data: the

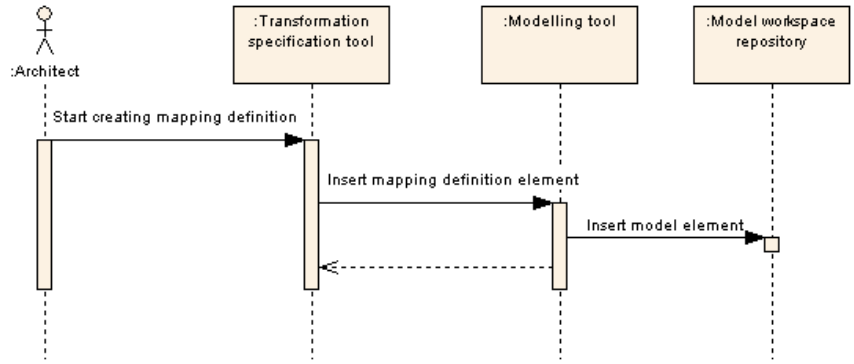

Fig. 2.20. Sequence diagram showing Engine dynamics for the "Create automatic mapping definition" use case

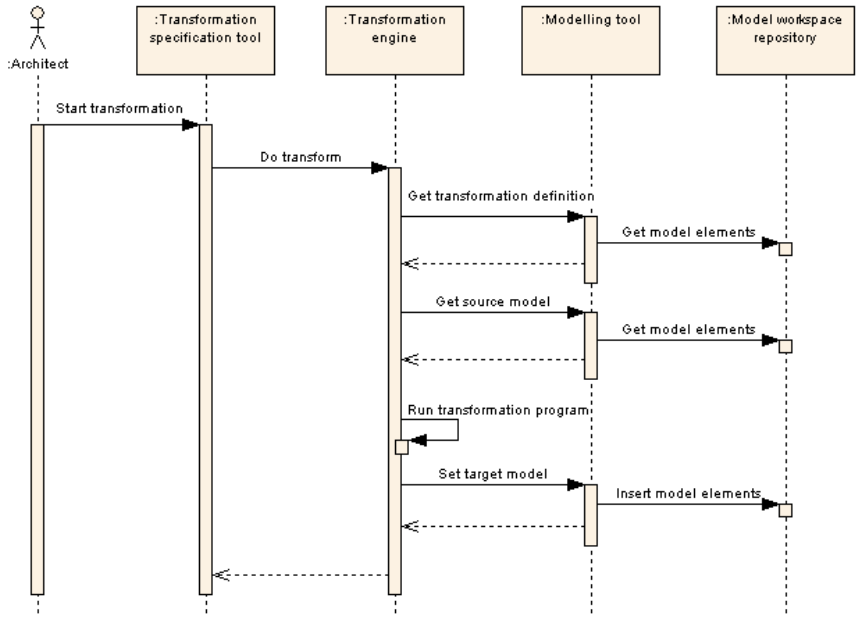

Fig. 2.21. Sequence diagram showing Engine dynamics for the "Transform requirements to architecture" use case

transformation definition ('Get transformation definition') and the source requirements model ('Get source model'). After fetching these two elements, the transformation program is run based on the transformation definition ('Run transformation program'). During this, the resulting model is constructed. Fi-

nally, the target model is stored in the repository in the chosen package ('Set target model'). Together with the target model, mapping links to the source model are also stored.

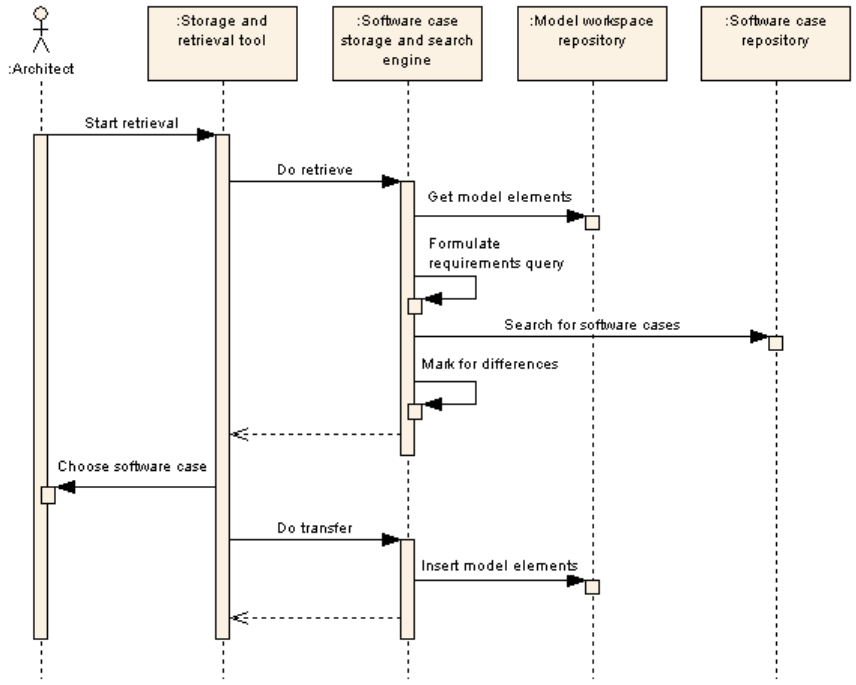

Fig. 2.22. Sequence diagram showing Engine dynamics for the "Retrieve and show modifications necessary in architectural model" use case

Retrieve and show modifications necessary in architectural model. (see diagram in Figure 2.22)
When the Architect wants to retrieve stored architectural models, he/she asks the `Storage and retrieval tool` ('Start retrieval'). The tool evokes the retrieval process performed by the `Software case storage and search engine` ('Do retrieve'). The search engine at first gathers the requirements model to be the basis for formulating a query ('Get model elements'). Having the requirements model, the search engine automatically formulates a query in a language understandable by the `Software case repository` ('Formulate requirements query'). This query is used to search for relevant software cases in the repository ('Search for software cases'). During this, appropriate similarity metrics are determined and also retrieved.

After retrieving the software cases with similarity metrics, the `Software case storage and search engine` marks them for differences in comparison

to the current software case. The algorithm uses traces between the requirements models and architectural model in the retrieved cases. Whenever a difference occurs between the current requirements model, and the retrieved model, this difference is propagated through the links into the architectural model. Then, appropriately linked architectural elements are marked as necessary for modifications.

After this, the Architect can browse the marked models (this interaction is not shown on the diagram). Finally, the Architect can choose on of the retrieved models ('Choose software case'). This software case, in its architectural part, is then transferred to the current `Model workspace repository` ('Do transfer' and 'Insert model elements').

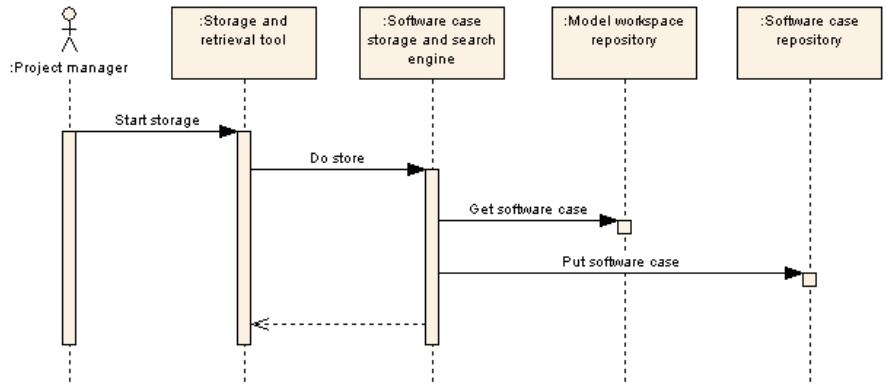

Fig. 2.23. Sequence diagram showing Engine dynamics for the "Store a software case" use case

Store a software case. (see diagram in Figure 2.23)
Whenever the Project manager judges the current software case is stable, he/she can store it through the `Storage and retrieval tool` ('Start storage'). The process is quite simple as it consists in retrieving the complete software case from the `Model workspace repository` ('Get software case') and saving it into the `Software case repository` ('Put software case'). It has to be noted that this process does not involve any transformations of the software case. The current software case already should have necessary information for easy retrieval in the future.

3

Building coherent software cases in a unified language

3.1 Case specification language in practice

In the previous Chapter, in section 2.2.3 we have introduced fundamental elements to be found in a coherent software case specification language. There, appropriate metaclasses express the types of elements to be specified by software developers. These metaclasses form the abstract syntax of the language. Software developers, though, need to prepare their specifications in some concrete (specific) syntax.

According to the methodology presented in the previous Chapter we divide the description into four parts, each of the parts describing models to be prepared in four phases of the software development process.

In the following sections we shall present elements of the concrete syntax of the language in a less formal way. This concrete syntax shall be formalised with a detailed metamodel in the next Chapter.

We shall follow the European Space Agency (ESA) standard PSS-05-0 [61], where generally, the software development process is divided into User Requirements Specification, Software Requirements Specification, Architectural Design and Detailed Design.

3.2 Writing user requirements

User requirements form the top layers of the so called "requirements pyramid". They determine the vision and scope of the potential system to be built but they do not specify any details. Generally, the user requirements are there to show the client what he/she might expect, and to allow for verifying the realisation of these expectations. The requirements pyramid is presented in Figure 3.1. The system vision is a specification of general features of the system in close connection with the business needs of the client. The system scope is a specification of the system necessary to determine its size and amount of effort needed to build it.

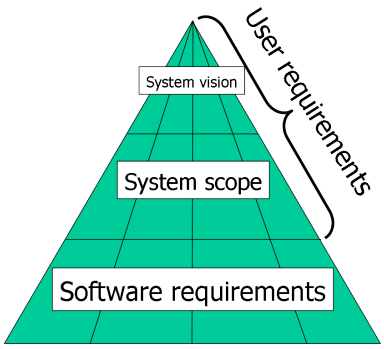

Fig. 3.1. Requirements pyramid containing user requirements and software requirements

Often, user requirements are written with unstructured common prose. However, this usually does not ensure proper quality meaning completeness, consistency, measurability (testability) and other important features. For this reason, requirements specifiers should classify their requirements into several types ensuring proper structure of a requirements specification.

- **Functional requirements** determine the system's behaviour while it is interacting with the user (or other system). They answer to important questions asked by the users. What services should the system offer? How should it react to specific input messages? How to behave in specific situations? The set of functional requirements determines the scope of the system to be built. Often it is wise to state also what functionality is out of scope (out of the system's functionality).
- **Vocabulary requirements** define the scope of notions and associated data to be handled by the software system. The vocabulary contains definitions of notions used inside functional requirements and quality requirements. When defining notions we also describe relationships between them. We can show these relationships graphically.
- **Quality requirements** describe the quality features of the prospective system. How fast should be the system? How reliable should be the system? How safe should be the system? How user-friendly should be the system? What norms should the system comply to? Non-functional requirements can be global (pertain to the whole system) or local (directly pertain only to specific functional requirements).
- **Constraint requirements** are determined by the business and technical environment that surrounds the system. They describe the external conditions set for the system (software, hardware, work conditions). Examples of constraints are local area network configuration, performance of client and server machines, database management system owned, working

environment (in the field, in a factory), or administrative personnel (how many people).

It can be noted that the requirements meta-model already presented in section 2.2.3 (see Fig. 2.7) satisfies this classification. There we have Notions, Units of functionality and Stories. These reflect two types of requirements: vocabulary and functional (respectively). In this book we shall concentrate only on these two types.

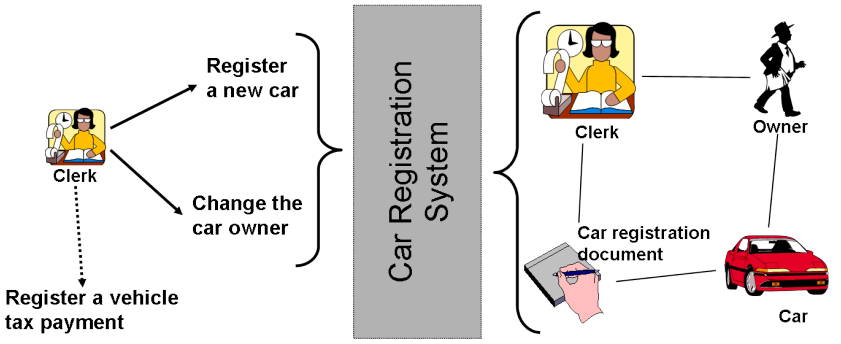

Fig. 3.2. Example of functional (left) and vocabulary (right) requirements describing visually two aspects of the same system

It has to be stressed that functional and vocabulary requirements should form a complete and coherent whole. These two aspects of the same system are orthogonal, but have very tight links. Inside functional requirements we use certain notions that should be present and defined in the vocabulary. This is illustrated for an example system in Figure 3.2. Requirements in this example are represented visually which gives an excellent overview of the system's scope.

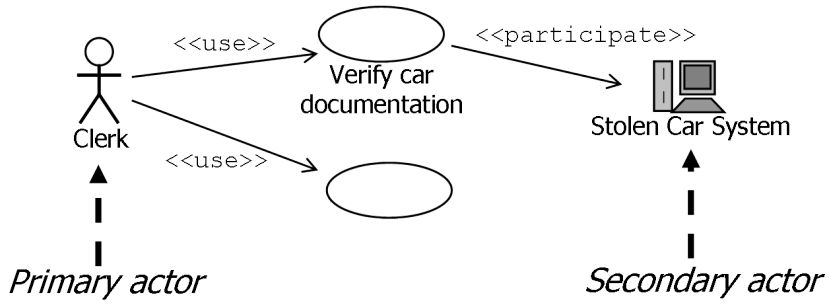

Fig. 3.3. Introducing the use case model with actor - use case relationships

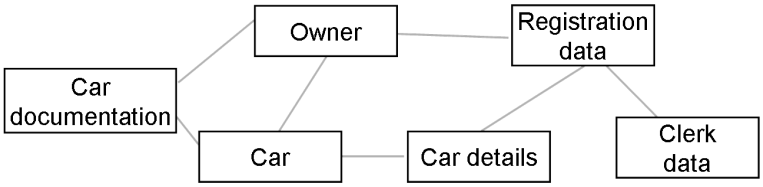

Fig. 3.4. Introducing the domain element model with notion relationships

The above example introduces the need for some concrete syntax of user requirements. This would allow for specifying them in a uniform way which gives potential for reuse as postulated in the previous Chapter. Here we shall propose syntax which is based on use cases and UML class diagrams. User requirements should thus contain two types of diagrams: use case diagrams (see Fig. 3.3) and domain element diagrams (see Fig. 3.4). These two types of diagrams reflect two models that are based on two main meta-classes shown in Figure 2.7). The use case model is composed mainly of Units of functionality in the form of use cases. The domain model is based on Notions. More details on the appropriate meta-models is given in the next Chapter.

- Use case model is composed of use cases and actors. Actors denote "roles" of people or machines outside the described system. Use case is a small piece of the system's functionality that: 1) begins with actor's interaction with the system, 2) describes the system's "dialogue" with the actor, and 3) leads to a specific goal that has a value to the actor. Use case model shows relationships between actors and use cases.
- Domain element model is composed mainly of notions and relationships between them. Notions denote data that shall be handled by the system. Every notion can define a packet of several data elements (attributes). Notion descriptions include also ways to process this data. It is important to capture relationships between notions. These relationships denote to us that one notion can be described in terms of another notion. It has to be stressed that notions are not UML classes, although notationally they resemble classes. So, notions do not have operations or attributes. Instead they can contain related phrases (see Section 3.3).

These two models enable us to describe the behaviour and data exchanged in a system composed of the "software system to be built" and actors interacting with it. This is illustrated in Figure 3.5. A typical use case is a sequence of messages between the actor and the software system. Within these messages, data is exchanged, where the structure of this data is explained through notions found in the vocabulary model. This gives us a coherent specification of the system's scope.

Actor describes a role that someone or something can play in respect to the considered system. Actor represents a group of people (or machines) that

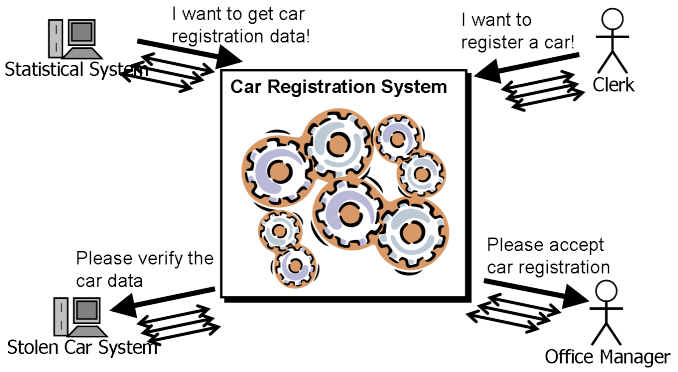

Fig. 3.5. Exchange of messages in a composite system consisting of actors and the system to be built

"talk" to our system in the same way. A particular person (or machine) can play several roles – can be associated with several actors. Actors communicate with the system by evoking use cases. A use case is a description of behaviour of a system communicating with one or more actors. In order for a behaviour description to constitute a use case three conditions have to be met:

- The description should start with an actor's interaction with the system,
- The description should present messages exchanged between the system and an actor,
- The description should clearly state the final goal reached at the end of message passing.

It is important to note that when describing a use case, we should treat the modelled system to be built as a "black box". Use case descriptions should concentrate on the behaviour that is visible to the actors. All the "machinery" inside the system should be hidden. The system is described as a "white box" only when we start to design it's architecture. Use cases with human actors describe only the user interface and certain business effects of the system's behaviour.

From the point of view of concrete syntax, relationship between an actor and a use case is denoted with an association (a line). Two types of relationships are possible:

- Primary actor for a use case: the actor starts the use case and the system tries to reach the use case's goal for this actor
- Secondary actor: the system asks the actor to help in reaching the use case's goal; communication with the secondary actor is done before the use case ends (during communication with the primary actor)

Primary and secondary relationships are denoted with appropriate arrows, as shown in Figure 3.3.

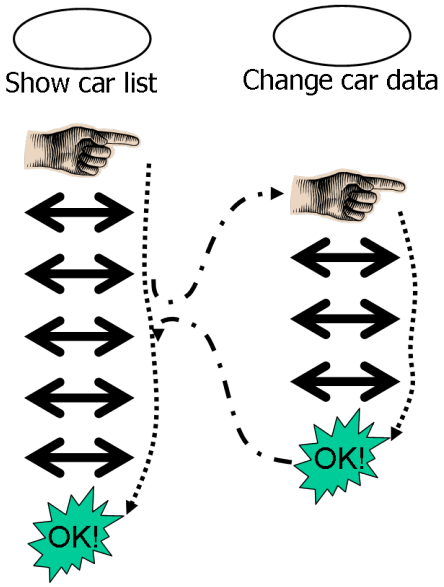

Fig. 3.6. Invoking a use case from within another use case

In addition to denoting relationships between actors and use cases, sometimes it is necessary to show that during execution of a use case, some other use case could be executed. Execution of another use case can be caused by user intervention, evoked by some condition of the system or performed unconditionally. All the interactions of the evoked use case are inserted into the main use case. It should be stressed that the evoked use case is a "normal" use case complying with the definition above. This is illustrated in Figure 3.6. The evoked use case could also be evoked independently of the evoking use case.

To denote relationships between use cases, UML has two types of invocation dependencies: «include» and «extend». These two dependencies cause certain level of confusion as their semantics described in the language reference is very ambiguous (see [186, 212, 132, 131]). For this reason we introduce a single dependency that substitutes both of the above: «invoke». Invoke means that interactions of the invoked use case are inserted (conditionally or unconditionally) into the invoking use case (as illustrated in Fig. 3.6).

Inside the use case description there can be several points where other («invoked») use case can be started. These points are called extension points. This is because they denote places where the main use case is extended with the functionality of other use cases. Extension point can be unconditional where the invoked use case is always inserted into the main one or conditional where the invoked use case is inserted under a certain condition. Conditions are shown in the example in Figure 3.7.

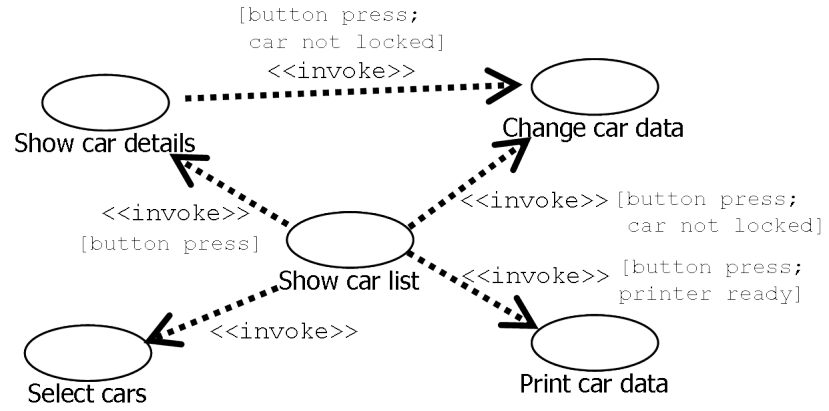

Fig. 3.7. Denoting «invoke» relationships

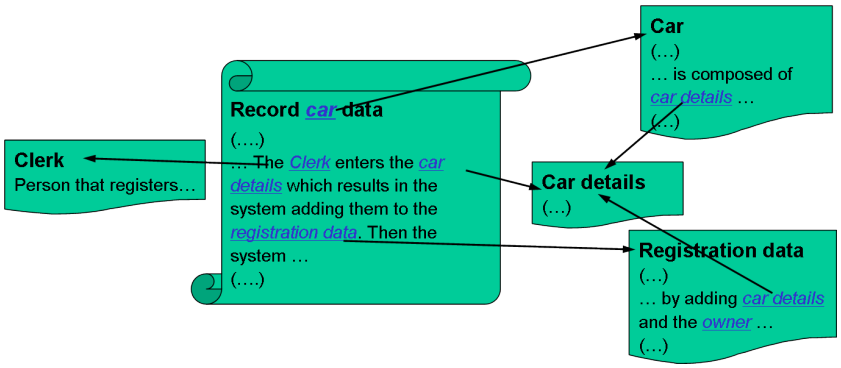

Fig. 3.8. Introducing hyperlinks to requirements descriptions

The use case diagrams contains important information about the system's functional scope. However, this is usually not enough for the project team to determine the size of the system to be built. Thus, a short description of each of the use cases is needed. It is within this description, and within the use case's name where the use case model intersects with the domain model. This is illustrated in Figure 3.8. It shows a short description of a single use case ("Record car data"). This description is hyperlinked to descriptions of appropriate domain elements (notions from the domain model, see [99] for an initial approach). It can be noted that these hyperlinks are reflected in the meta-model as an association between Notion and Unit of functionality in Figure 2.7. This gives us a very coherent and unambiguous requirements model. Certainly, keeping the hyperlinks synchronised is very tedious without

a specialised tool. Thus the need for the tool as described in the previous Chapter.

Defining use cases and domain elements determines the system's scope. All the use cases give us the scope for the system's functionality. All the notions define the scope for the data to be handled by the system. Scope of the system obviously determines its size. This gives us the means to make important cost and time estimates. Having the scope determined we need to specify the requirements in more detail, before developers start designing and coding the system. This is where software requirements come into scene.

3.3 Writing software requirements

Before designing the system we need to know more detailed information on the system's behaviour and its domain. This consists in extending the use case model and the domain model. Extending a use case means writing "stories" on how the system should behave (exchange information) in respect with the external objects (actors – users, other systems). Unfortunately, usually such stories are written more like a novel, where descriptions of the domain are buried inside the story. This leads to many inconsistencies. We often result in a situation where synonyms and homonyms are used leading to confusion. This leads to problems with translating requirements into design and then code.

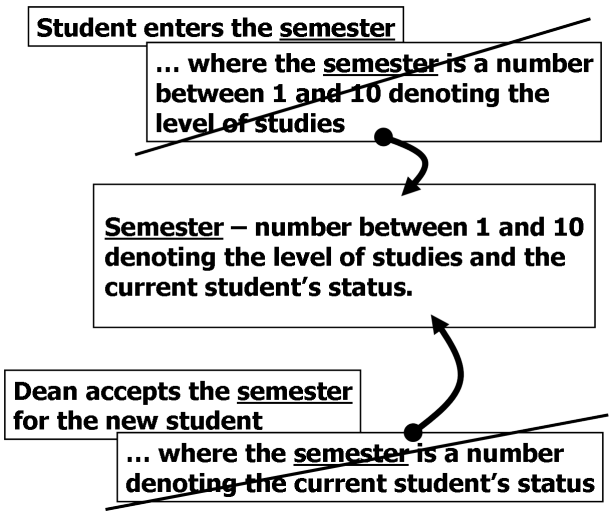

Fig. 3.9. Homonym problem in a software requirements specification

The "homonym" problem is illustrated in Figure 3.9. There we can see two story fragments, which both use the same term "semester". Unfortunately, these two terms are defined within the story, and they are both defined differently. This leads to serious confusion of developers in the future, where it is not clear which definition of the term "semester" is valid in a given situation. Keeping a separate and hyperlinked domain vocabulary solves this problem, as all the definition are kept outside of the story itself.

The proposed solution to the above problems is to specify a uniform format for writing scenarios. This format should unambiguously link scenario contents with the domain vocabulary. Moreover, this format should allow for easy translation into design constructs allowing for transformations as described in the previous Chapter.

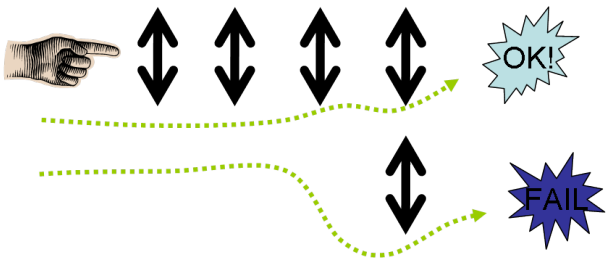

Fig. 3.10. Scenario with a triggering action, sequence of actions and goal.

When designing the format for scenarios we shall keep in mind their definition. A scenario is a sequence of actions forming a dialog between objects outside a system and that system. It is performed on behalf of a primary object ("primary actor") that triggers the scenario. The initial trigger is followed by a sequence of actions performed by the system and by the outside objects (including the primary object). The sequence is controlled by the system and leads to a single goal of significant value to the primary object. The sequence can fail to reach the goal – it is then a failure scenario. This is illustrated in Figure 3.10. It shows two scenarios with the same triggering action where one of the scenarios fails.

In order to specify the above sequence of actions it is enough to use the simplest possible sentences. Such sentences consist of a subject, a verb and one or two objects (here "object" meaning part of a sentence). With this SVO[O] structure of a sentence one can express an action performed by the system or the actor on one or two vocabulary elements. It can be noted that SVO[O] sentences do not allow for any definitions of domain elements. Appropriate example is shown in Figure 3.11. In these sentences, the subject is always the actor's name ("teacher", "admin") or the "system". Objects reflect notions to

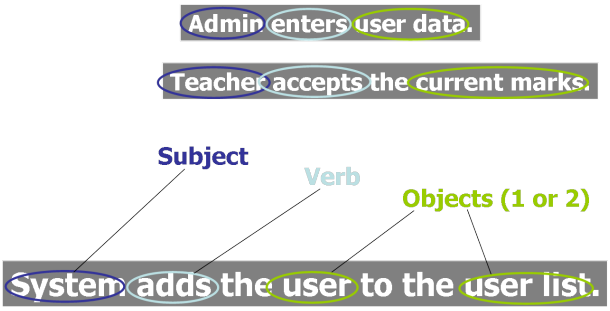

Fig. 3.11. Scenario sentences written in SVO[O] notation

be found in the problem domain ("user data", "current marks", "user", "user list").

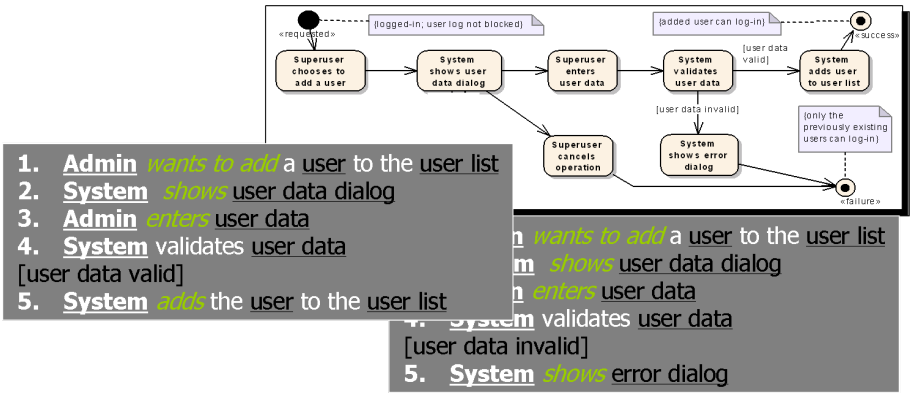

Fig. 3.12. Example scenario for a use case

Scenario can be written as a sequence of SVO[O] sentences. It is illustrated in Figure 3.12. The sequence always starts with the appropriate actor triggering an action. In the Figure we can see two scenarios that start with the same action ("Admin wants to add a user to the user list"). Then, there are two sequences of sentences. The difference between both scenarios lies in two conditions ("user data valid" vs. "user data invalid") which control the flow of events. The same two scenarios and a relationship between them can be also represented visually as an UML activity diagram. It can be noted that the diagram is equivalent to two textual scenarios. This diagram actually extends the textual version with another scenario which forms the third path through the activity diagram.

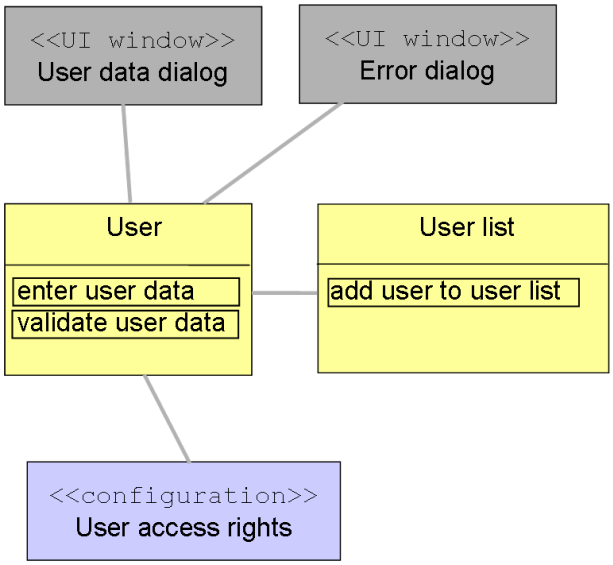

Fig. 3.13. Detailed domain model with notions and phrases

In the illustrated scenarios we can easily identify domain notions ("user", "user list"). These notions should already exist in the domain model created in the User Requirements phase. However, certain new notions, associated with the system's user interface appear ("user data dialog", "error dialog"). In other scenarios, notions associated with system parameters can also be introduced (see [219] on introducing new notions in the later stages of requirements specification). This necessitates extension of the domain model, as illustrated in Figure 3.13. Newly introduced notions in this model have appropriate stereotypes («UI Window» and «configuration») that allow to distinguish them from typical domain elements. The Figure also introduces certain phrases associated with notions. We can notice that for instance the User notion contains two phrases: *enter user data* and validate user data. The User list notion also has an appropriate phrase, and we could also add appropriate phases to other notions.

The phrases in Figure 3.13 can be compared to sentences in Figure 3.12. As we can see, these phrases exactly reflect verb-object-object parts of these sentences. In other words, the predicate (VO-O) part of any SVO[O] sentence can point to an appropriate phrase contained in a notion. This means that SVO[O] sentences denote in fact execution of sequences of phrases, as illustrated in Figure 3.14. This Figure can be compared with Figure 3.12. It reflects the success scenario as it passes through consecutive notions "evoking" their phrases. Numbers reflect numbering of the scenario sentences.

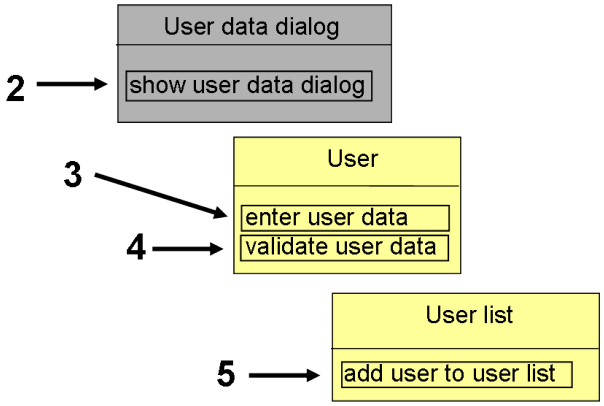

Fig. 3.14. Steps of an example scenario through the domain model

As we can see, software requirements update user requirements with additional details. The resulting model is still kept coherent while the domain model is consistently referenced (hyperlinked) within use case scenarios. Now, having detailed requirements for selected use cases, the developers can start designing the system.

3.4 Designing the architecture

Before going into the details of software design, including code elements, the developers should design the overall architecture of the system. Architectural level is necessary in order to be able to understand the system as a whole. This is due to inherent complexity of software systems with thousands of code modules to be designed and then implemented. We also need to handle somehow the complexity of translating requirements (which are usually also complex by themselves) to system implementation. Architecture gives a necessary level of abstraction that allows to see the bigger picture instead of concentrating on thousands of details.

The problem with designing software architectures is that there is no common understanding of what architecture is. The Software Engineering Institute gives over 30 definitions found in the literature (see http://www.sei.cmu.edu/architecture/definitions.html). There seems to be no commonly approved "language" to denote software architectures which is in a sheer contrast with standard blueprint format for building architectures. For this reason it is often that systems are being built without this necessary architectural level.

Here we shall assume the following characteristics that the software architecture should have:

- Architectural plan (blueprint) for a software system is an articulated set of decisions made by an architect. Architectural decisions are made on the basis of client's requirements and architect's knowledge about software development technologies.
- Architectural blueprints show the structure and dynamics of the system to be built.
- Architectural blueprint takes into account economical and technological constraints, at the same time making it possible to introduce changes and extensions to the initial requirements.
- Architectural blueprint is written in a graphical (visual) language understandable by the developers (programmers and designers of the software system).

Software architecture is an overall visual model of a software system, and architects should draw such visual plans. These plans should not reveal certain details of the system, thus only "top level" elements should be presented. These elements can be called components and appropriate architectural style can be called component architecture which we will present throughout this and the next Chapter. Component architectures have the following characteristics:

- A component architecture is an architecture where the system is divided into several (usually: in teens or in tens) logical modules which are individually responsible for a precisely defined functionality.
- A *component* is a "black box" that contains several smaller functional elements (e.g. classes in code) not visible to the outside world.
- A component makes the functionality of its contents (a set of public operations) available to the outside world through *interfaces*.
- When necessary, a component can utilise the functionality of other components by using their interfaces (through *dependencies*).

There are several arguments in favour of components.

- Component architectures make it easier to understand the overall system by supplying readers with good abstractions (components). Readers have an instant overview which can be quickly "exploded" into a detailed view, as the components are opened for closer inspection of their detailed design.
- Components enable better group work by dividing work very precisely. Groups concentrate on their own components and interfaces they use.
- Components make systems more flexible to change. It is quite easy to introduce new components and distribute components to different machines.
- Component architectures prevent from "spaghetti code" as communication paths between components are realised only through specified interfaces.
- Components make it easier to detect errors during testing. As independent black boxes, components are more resistant to errors in other components. Components are thus good for detecting errors as constituting "error centres" where errors can be localised.

- Components enable high levels of code reuse. Well designed and written components are ideal entities for reuse, which is important in the context o this book.

Components are high-level logical functional units of a software system. Every component has properties of a package as it contains other elements (e.g. classes). At the same time, component presents some behaviour. This behaviour, realised by certain code hidden inside the "black box" is available to the users of the system or to other components. Components expose their behaviour through ports. Ports are public interaction points between a component object and other objects. Detailed ways of communication between components are described with interfaces exposed from ports.

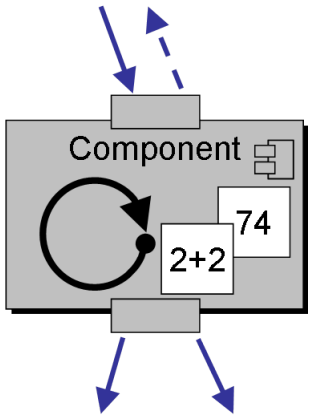

Fig. 3.15. Component exposing its behaviour through ports

A component with ports is illustrated in Figure 3.15. The actual notation for components is simple, as they are denoted as rectangles (with a small icon in the top right corner) with protruding ports (small rectangles). The Figure shows also the behavioural characteristics of a component. It sends and receives messages to the "outside world" through ports. A component starts behaving when it receives a message from the outside. The message instructs the component to "do something". After receiving a message, the component starts processing, it might do one or more of the following actions:

- Change its state,
- Perform some calculations,
- Send messages to other components asking for further processing.

When processing finishes, the component might return a result or just pass back control. This is valid for synchronous messages - asynchronous messages do not end with any return message.

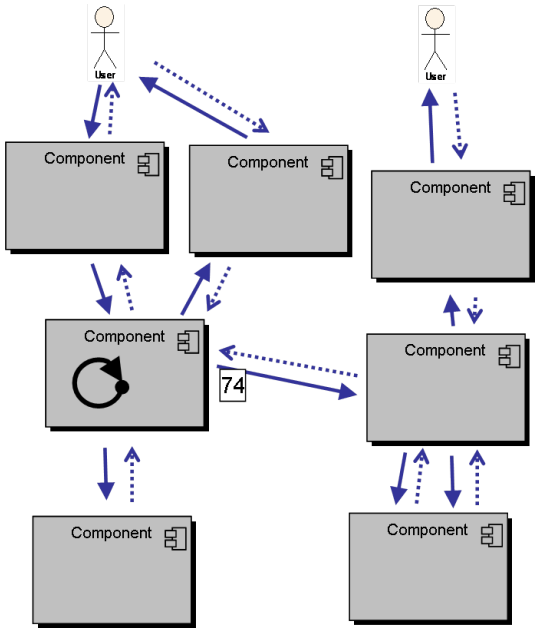

Fig. 3.16. Software system composed of components interacting through message passing

The behaviour of a component software system is based on interactions between components and users. Components interact by sending messages to each other which is illustrated in Figure 3.16. Sending a message usually means passing control (processing) from one component to another. Messages normally contain some information (data) that is passed. The overall system dynamics is a sequence of messages.

The above described behaviour of components can be defined more precisely by defining the interfaces they provide (realise) and interfaces they require (use). In Figure 3.17 we can see a component with associated interfaces. One interface is provided by the component (ICarRegistration) and other two interfaces are required from other components. The diagram to the left is a component diagram showing an overview, and the diagram to the right is a class diagram showing also the details of the interfaces. The interface definitions contain lists of operations with possible signatures (lists of parameters - not shown in the Figure).

Messages are elements of system's dynamics (dynamic architecture). Operations are elements of the system's structure (static architecture). Messages are passed between two objects (user objects or component objects). Messages can have parameters and return values. Whenever there is an interface involved in message passing, the set of available messages is limited to those

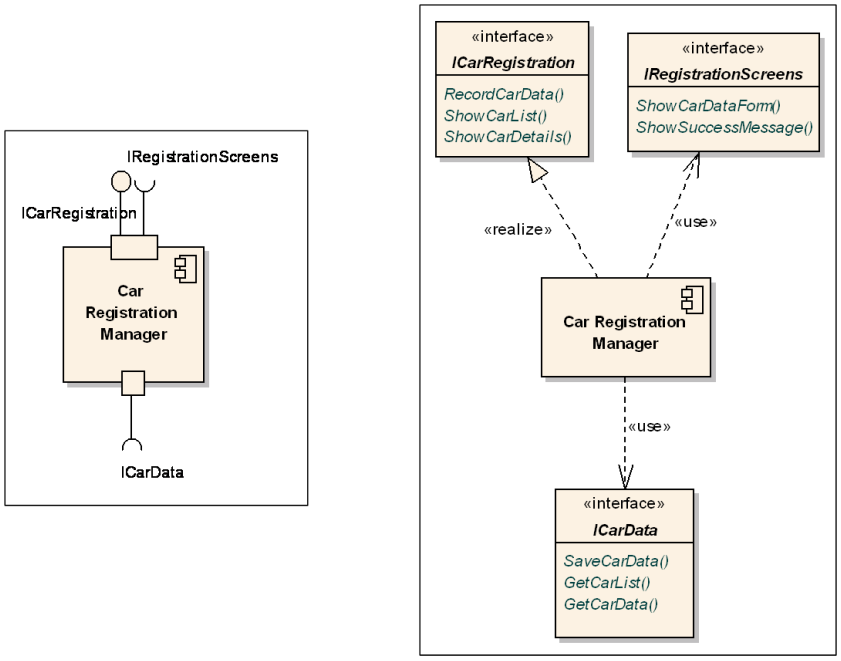

Fig. 3.17. Component's interfaces

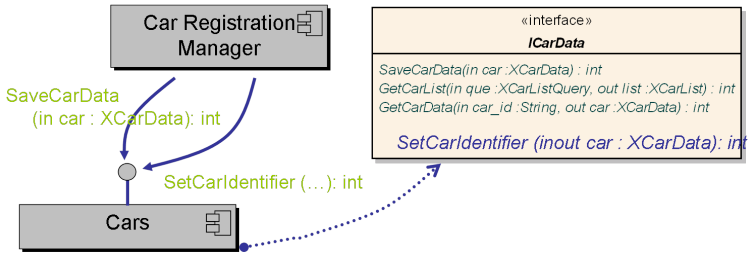

Fig. 3.18. Messages vs. operations - keeping coherence

defined in an interface. Whenever we want to add a new message sent between components we need to add new operation into the interface. This is illustrated in Figure 3.18. A new message (*SetCarIdentifier*) should be passed through an interface. This means that the appropriate interface (compare Fig. 3.17) should be updated with a definition of a new operation.

The above elements of the architectural model reflect the meta-model already defined in the previous Chapter (see Fig. 2.8). Components constitute Subsystems with Interfaces. These interfaces contain Services in the form of operations with parameters (equivalent to Data packets). These elements of

the Static architecture are tightly related with elements of the Dynamic architecture - Objects and Messages. The only element of the meta-model not yet introduced in this Chapter is the Story realisation.

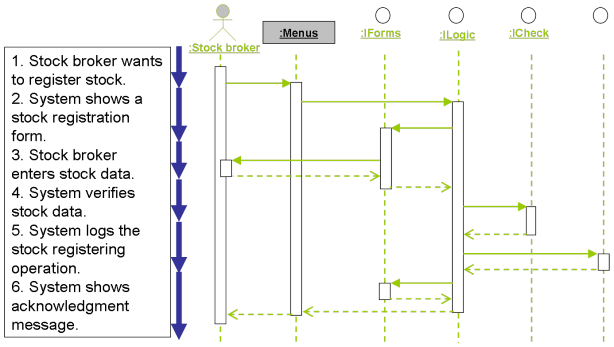

Fig. 3.19. Scenario as a sequence diagram

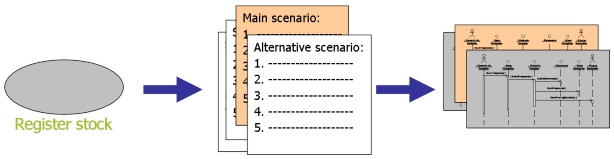

Fig. 3.20. Use Case realisation

Story realisations form the link between the requirements model and the architectural model. For every use case in the requirements model we need to specify the dynamics as a set of messages passing between components through their interfaces. We can take use case scenarios and for every SVO[O] sentence - design a sequence of messages revealing the "internals" of the system. It can be noted that designing the architecture means showing the internals of the "black box" specified by the use case model. This is illustrated in Figure 3.19. An example scenario is shown with an associated sequence diagram showing interface objects and a sequence of messages passing between them. A use case realisation is then a set of sequence diagrams each realising a scenario of the use case, as shown in Figure 3.20.

Sequence diagrams contain the mapping links (see Fig. 2.10) that join the SVO[O] sentences from the requirements model with messages in the architectural model. This leads us to the important issue of mapping between models. The mappings glue together all the elements of a software cases. Many of the mappings can be generated automatically having precisely defined transfor-

mation rules. At this level, these rules should determine the way use cases with SVO[O] sentences and vocabulary elements should be mapped into components, interfaces and messages. We shall briefly describe these rules in this Chapter and more details will be given in the next.

Before we can define the transformation rules we need to set some well-formedness rules for the architectural model. We need some guidelines for forming components into full architectural specifications. Otherwise we would have too many degrees of freedom in translating the requirements into architecture. Thus we shall perform our mapping with an assumption about the architectural framework to be the target of the transformation. This reflects the Target template in Figure 2.10. We shall use the same approach as was used when designing the ReDseeDS Engine (see Section 2.3.2). The target architecture shall consist of four tiers: Presentation, Application, Business Logic and Data storage.

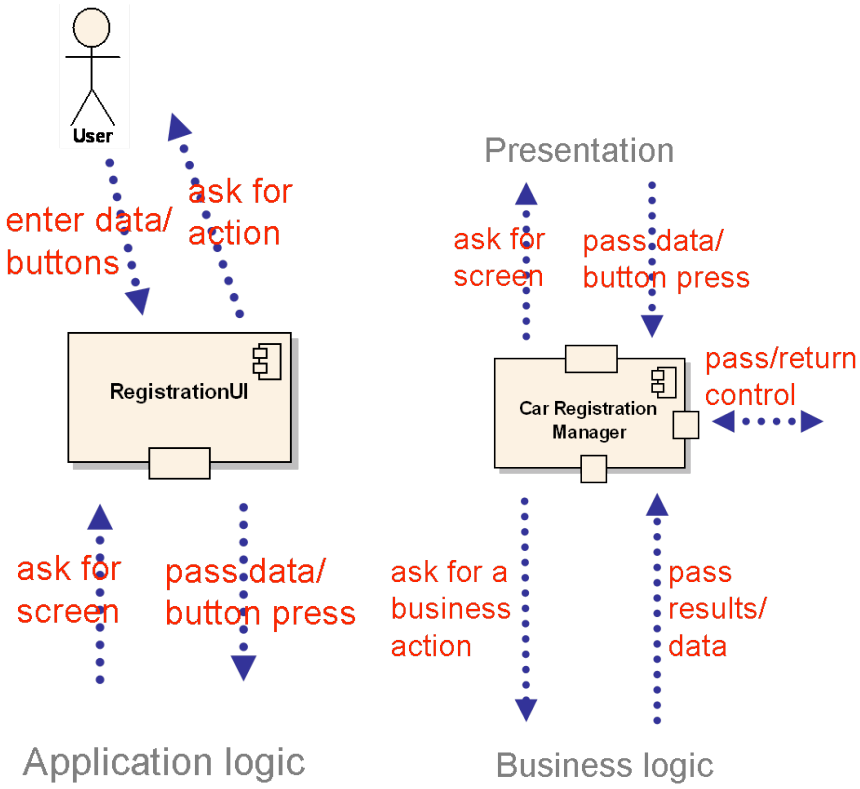

Fig. 3.21. Dynamics of the Presentation layer (left) and the Application layer (right).

Figure 3.21 shows the two upper layers. The Presentation layer serves presenting results and accepting data from the users. Components on this layer contain classes that handle individual screens. These classes can generate the screens and accept button presses. It has to be noted that in a four-tier architecture the presentation layer should be responsible only for presenting and accepting data from the user. No logic should be included here. It shows screens and accepts data only when instructed to do so by the Application layer or in response to user interventions.

The Application logic layer layer controls the whole application by asking other layers for actions in a specified sequence – according to the functional requirements. Components in this layer contain classes that manage appropriate sequences of events defined in use case scenarios. These classes know what to do when a certain button is pressed in a certain situation. They also react appropriately depending on the state of the system (asking the Business Logic). Application logic is the "manager" of the system, it instructs "everyone" (components in other layers) what to do and when.

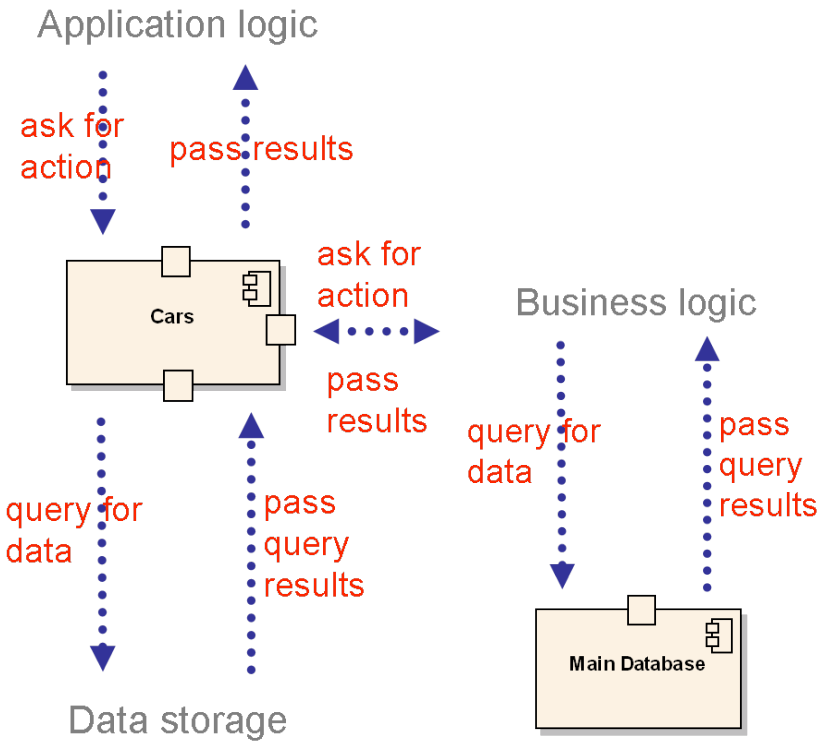

Fig. 3.22. Dynamics of the Business logic layer (left) and Data storage layer (right)

In Figure 3.22 we can see two remaining layers of the architectural framework. The Business logic layer performs data processing actions as asked by the Application logic. Actions performed by the business logic are consistent with the business rules defined in the vocabulary requirements. Components in this layer contain classes that represent business vocabulary notions. These classes have operations that process data contained in objects of these classes. Business logic is the right place to integrate with other applications. Different applications should integrate by asking for actions on this layer.

The Data storage layer allows for persistent storage and retrieval of data whenever it is needed by the business logic. Components in this layer contain databases, file stores, data repositories, knowledge bases etc. Relational database components (which are the most common) contain tables with relationships. Such components can handle queries in a query language (eg. SQL).

The above four layers should be properly "wired" through associations between interfaces and/or ports. This is illustrated in Figure 3.23 where all the provided interfaces are connected with appropriate required interfaces.

Having the above sketched framework for the target model we can now give brief rules for transforming requirements into design. In the next Chapter these rules will be presented in more detail. Generally, the transformation should be performed according to Figure 3.24. The uses cases should be grouped within the requirements model into packages. For every package of use cases, a component on the Application logic layer should be created. In case of «invoke» relationships between use cases assigned to different components, appropriate connections within the Application logic should be generated.

The above rule becomes obvious if we notice that the logic contained in the Application logic layer reflects use case scenarios. This layer is responsible for controlling other layers according to scenarios as specified within the functional requirements model.

Second major rule of the transformation is to package domain notions and for every such package create a component in the Business logic layer. In case of associations between notions assigned to different components, appropriate connections between components have to be made. Again, this rule becomes obvious when we consider that domain elements contain phrases describing actions performed within scenarios. These phrases reflect interface operations on the Business logic. It is the business logic that contains realisations of verb phrases defined within vocabulary notions (see Software requirements section).

Third major rule specifies the way the two layers should be connected. This should be made on the basis of hyperlinks between use case scenario elements and notions (or rather: phrases contained within these notions). Whenever such a hyperlink exists, a connection should be generated between the appropriate Application logic and Business logic components.

The final major rule describes the way the architectural sequence diagrams should be generated from use cases. Objects (lifelines) on these diagrams come

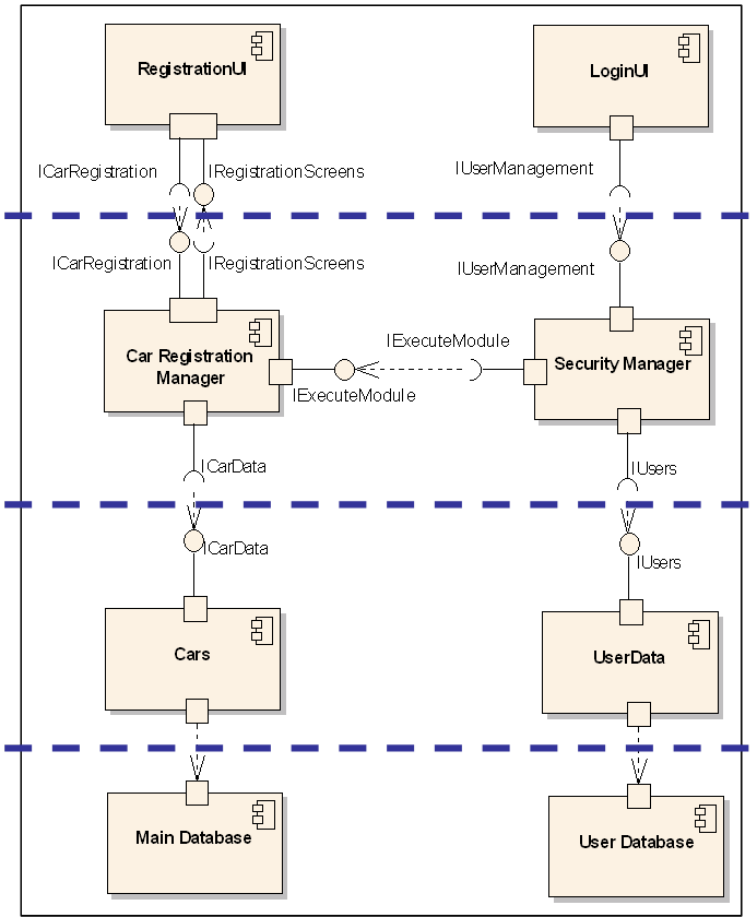

Fig. 3.23. Components connected in a four-tier architecture

from determining components that participate in a given sequence of SVO[O] sentences. Objects of appropriate components of the four layers are generated according to the association of use case and notions to these components made with the previous three rules. When this is determined, messages are generated by assigning SVO[O] sentences to source and target objects. This depends on the actual type of SVO[O] sentence. This and other rules shall be explained in much more detail in the next Chapter.

3.5 Designing the subsystems

Subsystem design is an activity analogous to architectural design. The difference which is reflected in the meta-model (see Fig. 2.9) lies in the fact

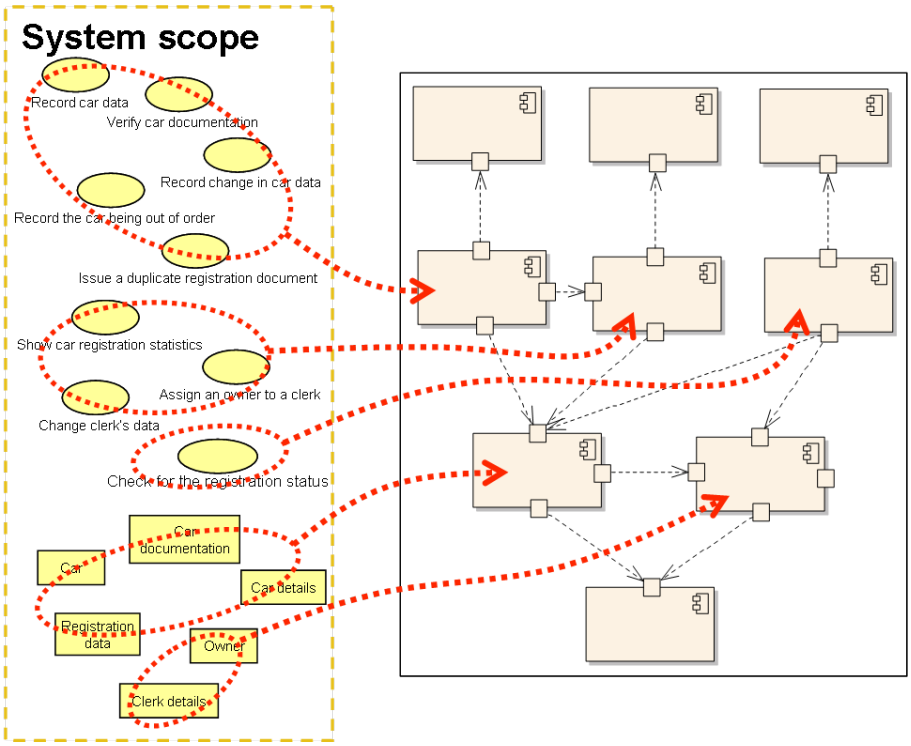

Fig. 3.24. From requirements to architecture

that at this level, operations (Services) get realised instead of use cases. Assuming object-oriented design is used, Code units are equivalent to classes (as present in any OO programming language like Java or C#) with operations and attributes.

Similarly as for the architecture, two aspects of the system need to be designed: its dynamics and its static structure. This time however we concentrate on interiors of components as illustrated in Figures 3.25, 3.26. We now need to "explode" components and operation executions associated with these components. Each of the components realises its provided interfaces. The static structure is a set of code units (usually - classes) contained within this component as shown in the first Figure. The dynamics of the component can be designed by showing message passing between objects of these code units. The sequence of messages, as shown in the second Figure describes the details of operation execution on the architectural level. An operation of an interface is now exploded into several messages showing the internal dynamics of the given component. It has to be stressed that the relationships between objects and messages on the dynamics detailed design sequence diagrams and static diagrams is analogous as in the architectural model.

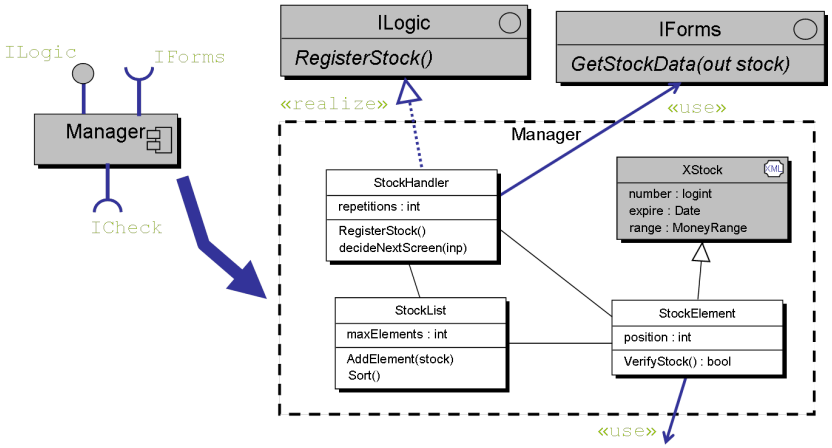

Fig. 3.25. Detailed design of a component consisting of several classes

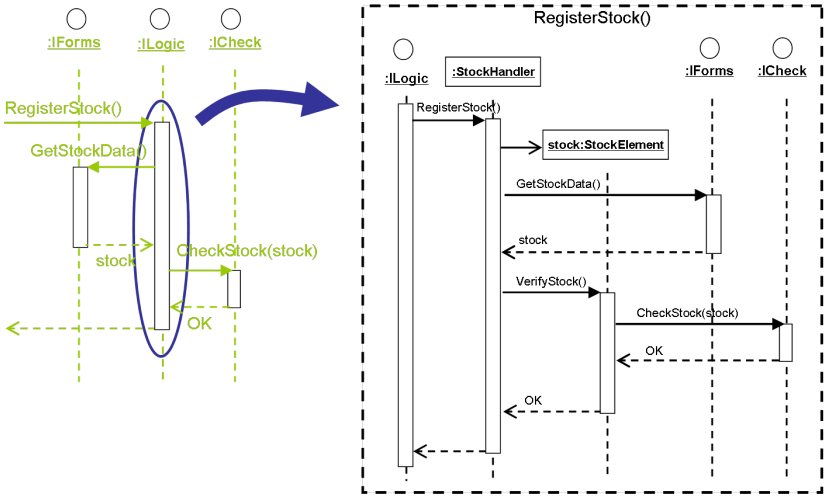

Fig. 3.26. Sequence diagram showing details of an interface operation execution

The above diagrams are already very close to code. In fact, their contents can serve generating code directly, as shown in Figure 3.27. The structure of code (classes, files) can be generated directly from design classes. The contents of class messages can be derived (although good automatic generators are not yet commercially available) from sequence diagrams. Close examination of the Figure shows that the code is actually equivalent (or is a combination) of both the static and dynamic aspect of the design model.

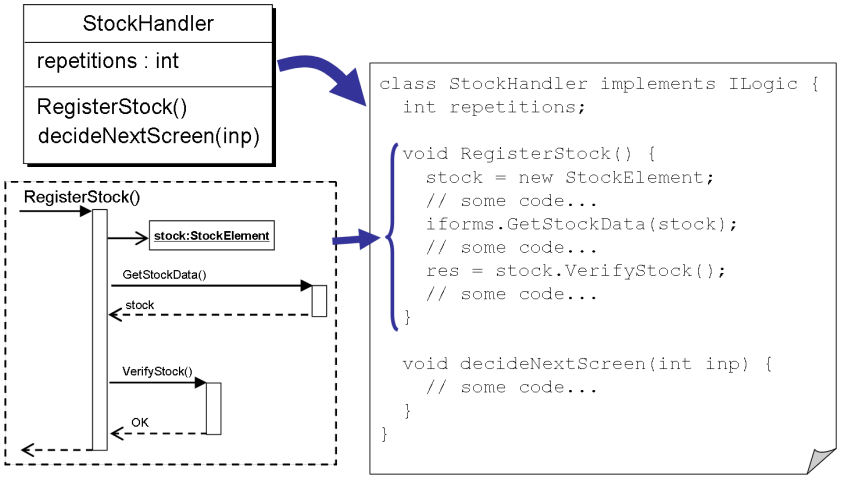

Fig. 3.27. Code generation from classes and sequence diagrams

3.6 Software cases and what next?

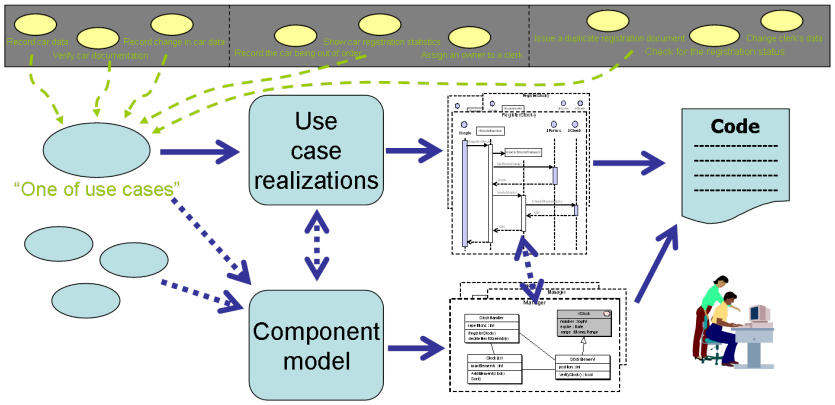

Fig. 3.28. Building a software case from use cases to code

The path from requirements to code which is presented informally in this Chapter leads to formulating software cases as postulated in the previous Chapter. Use cases are the basis for organising the software development lifecycle as shown in Figure 3.28. Together with domain notions they form the basis for generating and designing the architectural model. The architecture should be designed in detail by designing individual components and

the implementation of their interfaces. Both the static and dynamic aspect of software design gets finally reflected in code being combination of both.

The results of such software development activities can be stored in a library and retrieved for possible future reuse. In order for this scenario to become possible, more detailed considerations are needed. First, the requirements specification language has to be defined precisely (assuming we already have the definition of other parts of the software case, in the form of UML specification). Second, the way to define model transformations has to be determined and used uniformly. Finally, we need efficient ways to retrieve software cases. These together would allow for implementing the ReDSeeDS Engine and are discussed in more detail in the next Chapter.

4

Technologies for coherent and reusable software cases

4.1 Requirements modelling for reuse

As it was postulated in the previous Chapters, we need a uniform language for specifying requirements. With this language we should be able to define precise and unambiguous requirements specifications which would allow for easy transformation into design. When designing a language for requirements modelling we need to consider the existing general purpose languages as UML [154], SysML [205] or specialised languages as RDL (Requirements Description Language, see [35]) or RML (Requirements Modelling Language, see [86]).

The problem with UML is that its basic functional requirements units – use cases are defined in a very ambiguous manner (see [186] for good a discussion, still valid for UML2, and [72]). In SysML [205], the basic units of granularity are requirements themselves – their definitions being paragraphs of text. This is also the case for most widely used requirements management tools. Moreover, such models usually do not have precise links to the domain vocabulary. The vocabulary is often used inconsistently throughout the whole specification. This lack of common vocabulary also prevents from achieving any significant level of requirements reuse (see eg. [25] for a discussion on use case reuse). In turn, specialised requirements languages tend to be formal which prevents requirements specifiers from using them.

Here we shall provide more details of an approach to unify different tendencies in specifying requirements into a single coherent, yet comprehensible language. This language is supplied with a formally defined meta-model which allows for defining automatic transformations and comparison queries. This meta-model is consistent with the general idea presented in Section 2.2.3.

4.1.1 From natural language to models

As discussed earlier in this book we want to abandon the usual way of specifying requirements with paragraphs of common prose (natural language text).

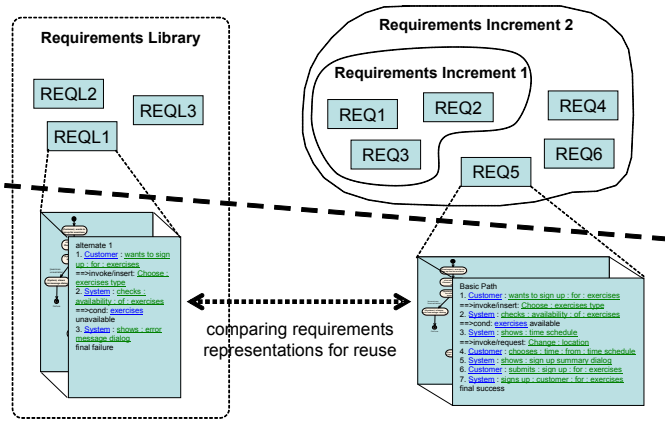

Fig. 4.1. Two levels of requirements management

This approach has several limitations that prevent from using such requirements as parts of software cases.

First limitation is caused by the fact that requirements as such are managed as wholes. This means that traces or mappings to or from requirements are done on the basis of "whole" requirements, not considering the requirement details (expressed through appropriate representations). Second limitation is based on inability to analyse the contents of requirements for comparison. Since the contents are paragraphs of text, the only way to compare requirements is through natural language processing which has serious limitations. This makes it practically impossible to compare different requirements specifications in order to achieve certain reuse of their contents (reuse of functionality descriptions, reuse of term descriptions and so on).

What can we do with "whole" requirements treated as atomic units? These requirements can be traced one onto another and can be assigned certain attributes. We can also trace certain requirements into design. Such functionality is offered by major Requirements Management tool vendors (RequisitePro, DOORS, etc.). Finally, some level of reuse can be achieved by attaching certain "manifestation" information to the requirement artifacts (as in Reusable Asset Specification, see [149]). It can be noted that maintaining traces or manifestation information is quite laborious and has to be done manually. Moreover we can assure that requirements are met in design only on a coarse grained level (whole requirements).

Thus, we need a requirements specification language where we can clearly distinguish between requirements as such and their representations (see [189] for an insight to this). Requirements as such are just names with identifiers and attributes. Representations express the information contained within requirements. This distinction introduces two levels of abstraction. On one level

we handle requirements as atomic units, on the other level we handle the details of their contents. "Requirements level" allows for coarse grained requirements management. "Requirement representations" level allows for automatic requirements processing (such as transformations and reuse). This is illustrated in Figure 4.1. On the upper level we can give order to requirements, eg. organise incremental delivery in an iterative lifecycle, assign requirements to people responsible for them or trace requirements into certain design artifacts. On the lower level we can perform fine-grained tasks of comparing requirements for reuse (see Section 4.3) and transforming requirements contents into design models (see Section 4.2).

Apart from functional and non-functional requirements, the requirements specification usually contains some form of a glossary or vocabulary of the problem domain. In certain approaches, also class diagrams are used to express the domain model. Diagrammatic approach is very valuable for the means of representing the domain vocabulary, due to its visual impact and expressiveness. However, using class diagrams in requirements specifications most often leads to design-influenced discussions. Moreover, class diagrams tend to concentrate on the "nouns" in requirements (classes and their attributes) rather than the "verbs" (operations, which are treated as design elements). When trying to define the verbs, analysts tend to fall into analysing (or rather: designing) possible message paths between objects. In other cases, "verbs" are completely ignored in the vocabulary and their definitions are scattered throughout the functional requirements definitions.

With the above approach we substitute purely natural language specifications with structured language and diagrammatic representations according to a specific formal grammar. Functional and non-functional requirements are made coherent through a central vocabulary of domain notions with associated phrases. These phrases use a globally recognised terminology which allows for comparison and leverage the potential for their reuse (see [28] for a domain based reuse approach with use cases). It can be noted that such a complete specification resembles a wikipedia, where hyperlinks are extensively used, as introduced by Kaindl (see [99]). Hyperlinks are organised around domain diagrams which introduce diagrammatic view of a vocabulary.

4.1.2 Unifying syntax for requirements

In order to reach repeatability between requirements specifications we need a common syntax for individual elements of such specifications. This syntax is part of a language which we will call the Requirements Specification Language (RSL, see [103]). The key concepts of RSL, like requirements, notions and requirements representations, can be expressed in a graphical way.

An RSL model, consistent with its syntax is called a RequirementsSpecification. Such specifications can be composed of RequirementsPackages. Within these packages, individual Requirements can be placed. Finally, every Requirement has its representations contained within it. RSL is mainly a visual lan-

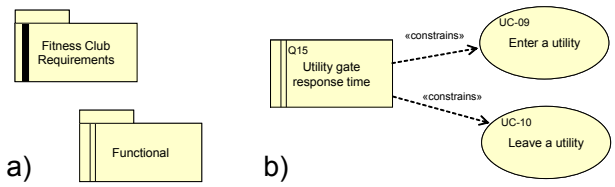

Fig. 4.2. RSL notation for specifications, packages (a) and requirements, use cases and relationships between requirements (example of one relationship type) (b) in diagrams

guage, thus appropriate elements written in this language can be shown in diagrams: Requirements diagrams, Domain diagrams, sequence and activity diagrams. Moreover, these elements can be shown as a hierarchy tree in a tool. Appropriate icons denoting packages are presented in Figure 4.2a. Distinguished from other Requirements (which can express any kind of requirement - functional, constraint, etc.) are UseCases, which specify "sets of actions performed by a system, which yield an observable result that is, typically, of value for one or more actors or other stakeholders of the system" (as specified in [154]). UseCases are denoted with an oval (see Figure 4.2b) in order to make this notation consistent with the commonly used UML notation and different from other Requirements. It can be noted that requirements, besides their descriptive name, have a unique identifier placed in their top left corner.

The above notation allows to depict individual requirements, what we still need is the way to express relationships between requirements and traces to design. In Figure 4.2b we can see also a notation for RequirementRelationships. This diagram shows relationships between one quality requirement and two use case requirements. Also, other types of relationships are possible, however we shall omit them here for brevity.

The most important part of the RSL is that pertaining RequirementRepresentations. In general, all the representations are based on the concept of hyperlinks. Even the natural language representations allow for including hyperlinks to domain elements or phrases. An example of such a natural language representation of a requirement from Figure 4.2b is presented below:

- `Whenever entering or leaving a [[n: utility]], the time between [[v: swiping n: card]] and [[v: opening n: utility gate]] should be smaller than 0.5 second.`
- Whenever entering of leaving a utility, the time between swiping : card and opening : utility gate should be smaller than 0.5 second.

In this example, PhraseHyperlinks to three phrases are used. Two of the phrases are simple noun Phrases, and one of them is a VerbPhrase. It can be noted that the RSL notation allows for a "source" version and a "preview" version of any textual requirement representation. In the source variant, every

element of a PhraseHyperlink is denoted by a prefix: *n:* for a noun expression and *v:* for a verb expression. In the preview variant, phrase parts are separated with a colon "*:*". In the following examples we shall only show the source versions. Hyperlinks can also contain modifiers (usually adjectives) but we shall omit them here for brevity.

In contrast to NaturalLanguageRepresentations described above, ConstrainedLanguageRepresentations have a precise grammar defined. In this section we shall concentrate on ScenarioSentences where SVOScenarioSentence is the most important of them. SVO[O] sentences [82], [83], [196] are simple sentences consisting of a subject and a predicate. This predicate is composed of a verb and and one or two objects (with an optional preposition). Examples of such sentences are given below:

- `[[n: FC System]] [[v: opens n: utility gate]]`
- `[[n: Customer]] [[v: swipes n: card]]`
- `[[n: Assistant]] [[v: chooses n: bill p: from n: bill list]]`
- `[[n: FC System]] [[v: prints n: bill]]`

The syntax for SVO[O] sentences allows for only hyperlinks. The first Hyperlink relates to a domain element representing the sentence subject. This domain element can be the system to be built or an actor (representing someone or something interacting from the outside of the system to be built). In the example above, the system to be built is "*FC System*" (Fitness Club System), and the actors are "*Receptionist*" and "*Customer*". The second PhraseHyperlink represents the sentence predicate. It relates to a VerbPhrase contained in a specific notion in the domain vocabulary.

We can combine sequences of such SVO[O] sentences into ConstrainedLanguageScenarios. These scenarios show interactions between an actor (or actors) and the system to be built. In other words they present a dialogue between the user and the system. Two example scenarios are shown below:

1. `[[n: Customer]] [[v: swipes n: card]]`
2. `[[n: FC System]] [[v: verifies n: account balance]]`
3. `==> cond [[n: Account balance]] not exceeded`
4. `[[n: FC System]] [[v: opens n: utility gate]]`
5. `[[n: Customer]] [[v: passes n: gate detector]]`
6. `[[n: FC System]] [[v: registers n: utility entry]]`
7. `[[n: FC System]] [[v: closes n: utility gate]]`

and

1. `[[n: Customer]] [[v: swipes n: card]]`
2. `[[n: FC System]] [[v: verifies n: account balance]]`
3. `==> cond [[n: Account balance]] exceeded`
4. `[[n: FC System]] [[v: emits n: rejection message]]`

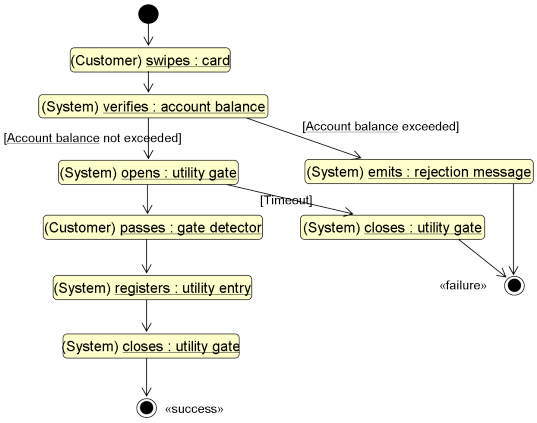

Fig. 4.3. Activity scenario expressed as an activity diagram – consistent with the constrained language scenarios

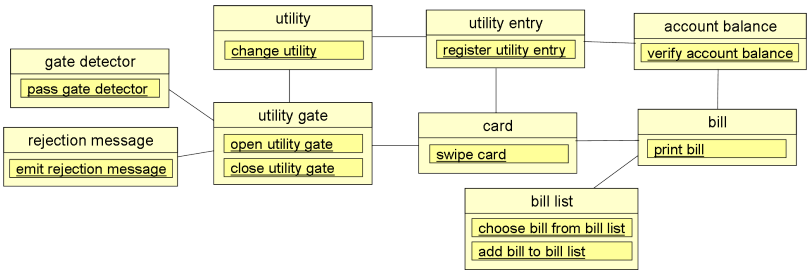

Fig. 4.4. Domain diagram showing notation for notions and phrases – targets for the hyperlinks

As it can be seen, the two scenarios complement each other and show two alternative paths of the same UseCase: "*Enter a utility*". They contain two conditions denoted by ==> *cond* which control the flow between these two paths. Both of the scenarios can be placed inside the "*Fitness Club Requirements*" specification (see Fig. 4.2) – within the *Functional* package under the appropriate use case. There we can also place another representation of the UseCase – in the form of an ActivityScenario. This representation is shown in Figure 4.3. The two representations contain almost exactly the same information. Additional notation for sentences in this diagram allows for distinguishing parts of SVO[O] sentences forming scenario steps. It can be noted that the activity diagram contains an additional path and distinguishes success and failure paths.

The domain model is the target of all the hyperlinks presented in the above examples. For a domain model we draw domain diagrams. These diagrams have certain elements in common with class diagrams, but they shift the paradigm from classes with operations to notions with phrases. Domain

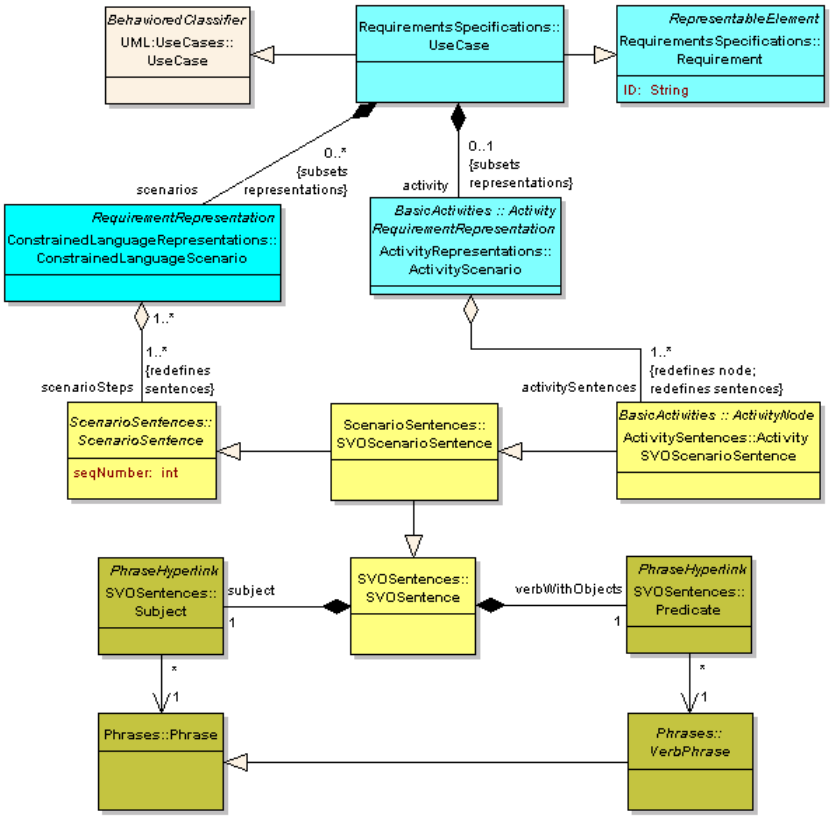

Fig. 4.5. Summary of the RSL metamodel

diagrams contain DomainElements (specifically: Notions) expressed as boxes. These domain elements (notions) can contain Phrases, where every phrase is also a box. This is shown in Figure 4.4. The diagram contains all the phrases hyperlinked within the presented example requirements.

Moreover, all the DomainElements are related with appropriate DomainElementRelationships. These relationships are created on the basis of hyperlinks in domain element descriptions. Let's take a description of the "*card*" notion: "... Groups all [[n: utility entries]]. Can be swiped through a [[n: utility gate]] ... for every card a [[n: bill]] can be issued ...". As it can be seen, the above hyperlinks lead to appropriate Notions related to the "*card*" notion as shown in Figure 4.4.

4.1.3 Meta-model of the Requirements Specification Language

Behind the above presented concrete syntax of the RSL, a meta-model defining the language's abstract syntax resides. This meta-model is shown in a very brief overview in Figure 4.5. This Figure presents most of the elements described in this section. We can see the Requirement meta-class which denotes the most important element of the language. Another basic element is the DomainElement (see Figure 4.8). Both of these elements are RepresentableElements, ie. elements that have associated Representations. Such 'representations' are shown for a UseCase meta-class which is a specialisation of Requirement. It can be seen that use cases can have as their representations ConstrainedLanguageScenarios and ActivityScenarios. It should be noted that use cases are specialised from UML UseCases (which are BehavioredClassifiers) and activity scenarios are specialised from UML Activities.

Below in Figure 4.5, we can see some selected elements of the structure of the two use case representations. Constrained language scenarios contain ScenarioSentences as their 'scenarioSteps'. Activity scenarios generally contain 'nodes' which can be (among other) ActivitySVOScenarioSentences. Most of the sentences in scenarios are SVOSentences (there can be also eg. ConditionalSentences). Such sentences contain a 'subject' and 'verbWithObjects' being its Predicate. These two sentence elements are Hyperlinks that point to a Phrase or a VerbPhrase respectively.

The meta-model for phrases is presented in Figure 4.6. Phrases are generally sequences of hyperlinks pointing to terms. These Terms (with their forms, inflections, cases) are stored in an external, global structure. This structure, represented by a Terminology (detailed discussion on Terminologies isout of scope of this book), contains relations between Terms as well. These relations define the semantics of the Terms. Such structure in fact is an ontology and can be based on existing dictionaries/ontologies (e.g. WordNet [64]). Moreover, it can be founded upon a set of syntactic rules and semantic elements taken from natural languages (as is described in [35]).

The simplest Phrase contains just the Object (hyperlink to a Noun). Such Phrase is adequate for defining names of Notions. Another simple Phrase can contain optionally a Modifier and a Determiner.

VerbPhrase describes an operation that can be performed in association with a Noun. VerbPhrase is an abstract subtype of Phrase and it exists in two concrete variants: SimpleVerbPhrase and ComplexVerbPhrase. SimpleVerbPhrase is the basic structure for expressing Noun's behaviour. In addition to a Phrase, it includes a PhraseVerb (hyperlink to a Verb). It may also contain a PhrasePreposition (hyperlink to a Preposition). ComplexVerbPhrase describes a behavioural relation between two Nouns. It extends the SimpleVerbPhrase with an additional Noun (indirect object) through an Object hyperlink. It is a kind of VerbPhrase pointing to a SimpleVerbPhrase. It also includes a PhrasePreposition.

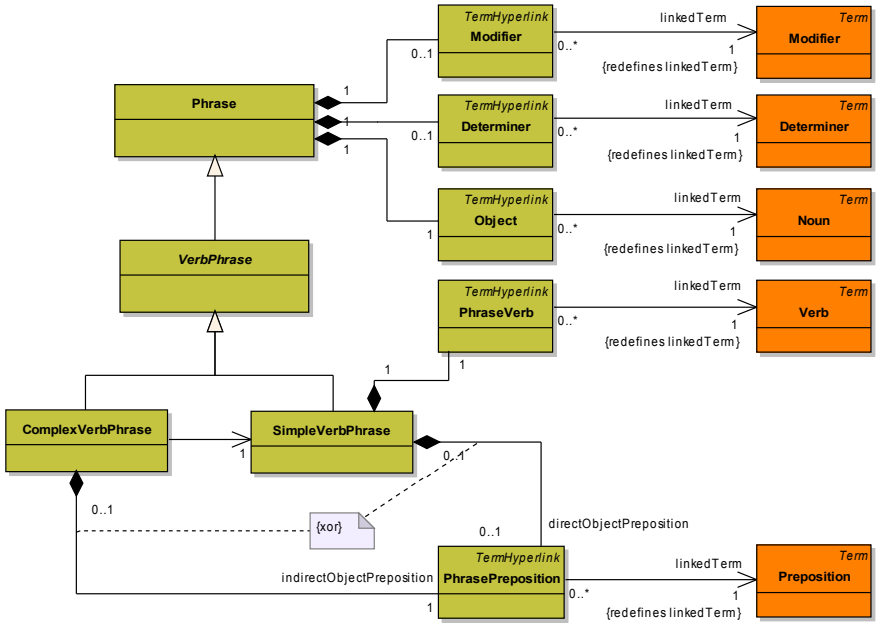

Fig. 4.6. Meta-model for phrases

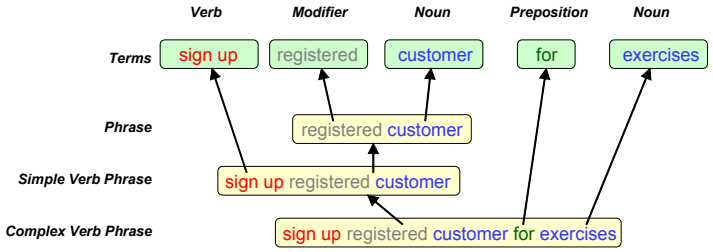

Fig. 4.7. Structure of phrases in concrete syntax

In Figure 4.7 we can see an explanation of the above meta-model in terms of a concrete example. The example shows a set of terms and several phrases built of these terms.

In the RSL meta-model every Phrase is tightly combined with the Noun being part of it. This Noun is the name of the Notion that contains this phrase, through a NounLink. A simplified meta-model for notions is presented in Figure 4.8.

Every Notion, which is a kind of UML :: Kernel :: Package, can include many so-called DomainStatements referring to the same noun. These DomainStatements can contain free text hyperlinked descriptions as discussed earlier. In its concrete syntax, a Notion contains an overview of all the Phrases included in

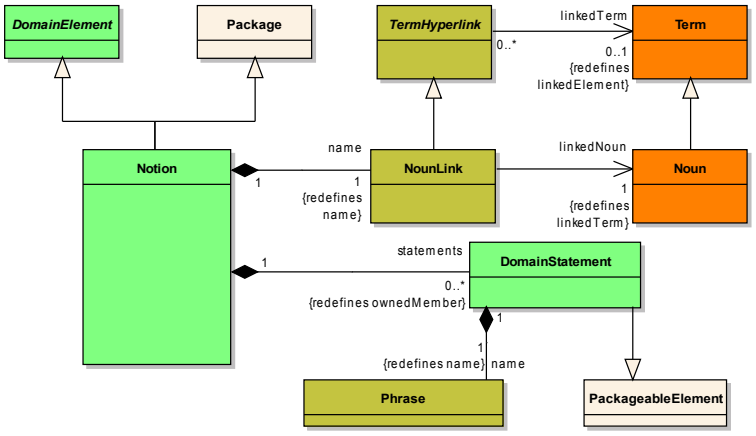

Fig. 4.8. Meta-model for notions

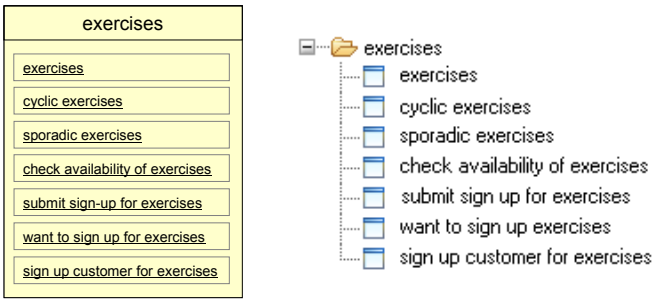

Fig. 4.9. Notions represented as nodes in domain diagrams (left) and as trees (right)

it. It can be presented as a domain element icon on a domain element diagram

Source:

Exercises

Form of physical activity performed in [[n:fitness club]]. Exercises may by [[m:cyclic n:exercises]] or [[m:sporadic n:exercises]]. [[m:Sporadic n:exercises]] must be reserved earlier (in eg. private session with [[n:coach]]). Cyclic exercises needs earlier registration.

[[m:cyclic n:exercises]]
[[m:sporadic n:exercises]]
[[v:check n:availability p:of n:exercises]]
[[v:submit n:sign-up p:for n:exercises]]
[[v:want to n:sign-up p:for n:exercises]]
[[v:sign-up n:customer p:for n:exercises]]

View:

Exercises

Form of physical activity performed in fitness club. Exercises may by cyclic : exercises or sporadic : exercises. Sporadic : exercises must be reserved earlier (in eg. private session with coach). Cyclic exercises needs earlier registration.

cyclic : exercises
sporadic : exercises
check : availability : of : exercises
Submit : sign-up : for : exercises
want to : sign-up : for : exercises
sign-up : customer : for : exercises

Fig. 4.10. Notion represented in textual form

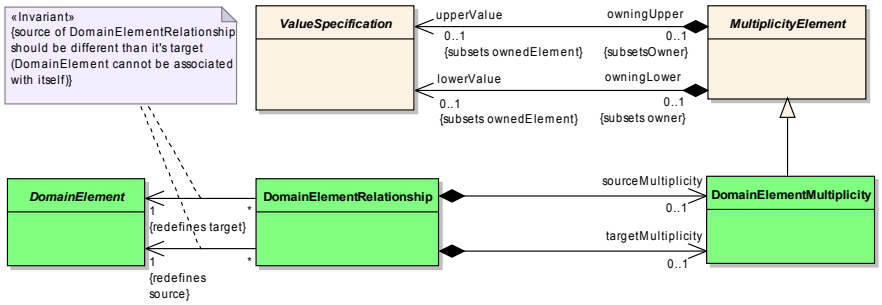

Fig. 4.11. DomainElement Multiplicities

or as a node in a tree view (see Figure 4.9). Notion can be also represented in a textual form with hyperlinks (see Figure 4.10).

To complete the domain model, we need to define the relationships between Notions (or: DomainElements, refer to Fig. 4.8). This is illustrated in Figure 4.11. Here we can see that DomainElements are related through a relationship (see specification of relationships in the UML specification, [154]) called DomainElementRelationship. This relationship can have multiplicities (DomainElementMultiplicity) with appropriate upper and lower bound, similarly to what can be found in UML class diagrams.

With the above meta-model we can now create domain specifications as shown eg. in Figure 4.4. It can be noted that through this meta-model we have achieved that the domain specification can be organised in the form of a dictionary with hyperlinks, as postulated in previous Chapters. Moreover, entries in this dictionary (notions and phrases) can be targets of hyperlinks from the requirements specification (inside SVO[O] sentences or hyperlinked descriptions).

Having the requirements specification language as described in this section we now have the means to achieve the postulates formulated earlier in this book. We can now define formal transformations of requirements into design and formulate queries seeking for similar requirements. This shall be described in the following two sections.

4.2 Defining complete cases with model transformations

The above presented requirements specification language can be part of the overall software case specification language as introduced in Section 2.2.3. The remaining parts of the language consist of architecture, detailed design and code. In the previous Chapter we have briefly described the method of building software cases gradually from requirements to code, containing all the properly mapped parts. According to the ideas of MDA/MDD it was proposed to use automatic transformations to support the developers in this

task (see eg. [73]). Here we shall elaborate on this issue by giving precise rules for transforming different level models. We shall also give an example of a language to define transformations which was introduced in Section 2.2.3 (see Fig. 2.10).

4.2.1 From RSL specifications, through UML models down to code

According to the software development scenarios presented in the previous Chapters we want to be able to generate as much code as possible from requirements. This does not mean purely automatic transformation. This could be possible only if no architectural or design decisions were necessary. Most often however we need to consider specific non-functional requirements which might significantly influence the architecture of the system and detailed design solutions. Since we did not include non-functional requirements in the transformation path, we need to allow for decisions reflected in design models modified manually by software developers. In any case we will show here that much of the work done traditionally by architects and designers can be transferred to "transformation modelers". Many architectural decisions can be done in the transformation definitions. These transformations can be used many times in different projects thus reducing design effort.

While designing rules for model transformation we shall assume a clean path leading from requirements to code. In this path, a requirements specification is first transformed into an architectural model. Then, this architectural model is elaborated through a detailed design model. Finally, code is generated form detailed design. It is important to assume that no "shortcut mappings" are introduced. Such shortcuts could for instance consist in taking requirements to influence the detailed design or code. In a clean software case, all the mappings are made only between two consecutive levels as shown in Figure 2.6. This means that all the information contained in requirements has to be somehow included in the architecture. During a transformation between requirements and architecture we should transform all the elements of the requirements model. Only then, there will be no need to take information from requirements while generating detailed design or code. Those elements that are not transformed automatically from requirements to architecture have to be manually included in the final architectural model before transforming it into detailed design and code.

When defining transformation rules we need to take certain assumptions on the architectural and detailed design models. Having these assumptions we can include them in the rules thus enabling automatic generation. Here we shall assume exactly the same architectural framework as presented in the previous Chapter. We shall assume a four-tier architecture depicted with a UML model consisting of the following elements (compare Fig. 2.8, refer to [106] for more details):

- static elements of the architecture: Components, Interfaces, Dependencies, Classes, Packages (component and class diagrams).

- Dynamic elements of the architecture: Lifelines, Messages (sequence diagrams).

This model can be updated by the architects by using additional UML element types, adding new elements and updating the existing ones. However, changes in the elements generated from requirements should be taken with care. Every such change might mean that a certain guideline for the architectural model is not kept anymore. This might also mean that a certain requirement stops being realised in the architecture. This situation can be compared to generating code from UML models. Every change in the generated code elements means that the design model becomes out of date. This then necessitates the usage of reverse engineering techniques (re-generating model from code) in order to retain consistency.

Now, having the architectural model we can transform it into detailed design. Again, certain assumptions on it have to be made in order to make this possible. For the purpose of this book we shall assume that detailed design shall be written in UML using only the following elements (compare Fig. 2.9):

- Static elements of subsystem design: Classes, Interfaces, Associations, Realisations, Dependencies (class diagrams).
- Dynamic elements of subsystem design: Lifelines, Messages, Combined fragments or Conditions (sequence diagrams).

While creating transformation rules for the detailed design we have to keep in mind the purpose of the four tiers. This purpose determines the contents of components contained in different tiers (classes) and their dynamics (messages exchanged within and between tiers).

The Presentation Tier contains basically the User Interface elements of the designed system. We can assume that a single UI component with a single interface is built for an application. This component serves only for mediating between the users and the system's logic. This means, no logic should be contained in this tier. It is common to implement certain simple validation logic, however we shall avoid this in our transformation rules.

The Application Logic Tier is responsible for realising the rules specified through the functional requirements. Thus, according to the previous Chapter, this tier contains components with logic reflecting use case scenarios. We shall assume that every use case has an associated interface which handles messages from the Presentation Tier (reflecting user input in the scenarios). Interfaces are realised by appropriate components which are generated on the basis of grouping of use cases. This grouping has to be made by the requirements specifiers.

The Business Logic Tier handles all the business rules contained in the requirements. Whenever the Application Logic needs to perform a business action, it asks the Business Logic to perform it. In turn, the Business Logic calls the Data Storage Tier to perform basic persistence operations. The Business logic layer consists of components, which correspond to related groups

of domain concepts - notions. Appropriate interfaces are created which reflect groups of operations on notions.

The Data Storage Tier constitutes direct access to a source of persistent data (a relational database or a flat file). Interfaces and components in this tier expose basic CRUD (Create/Read/Update/Delete) operations on data access objects (DAOs) reflecting every notion in the requirements vocabulary.

In the following two subsections we shall present detailed rules for transforming requirements written using RSL into the above four-tier architecture and then into detailed design for the components contained in these four tiers.

4.2.2 Rules for transforming requirements into architecture

When designing the rules for transforming requirements into architecture we need to consider several pre-requisites. We have to remember that the source model is a requirements specification written in a precise language (RSL). The target model should be a well formed draft of the architecture that conforms to best practices of architectural design. Thus, the transformation definition should:

- Allow for generating architectural model conforming to such good practices as proper granularity of components, good ratio of interfaces per component and methods per interface.
- Assure that changes in the source model have predictable impact on the resulting model thus ensuring good mapping between requirements and architecture.
- Assure flexibility in customising the transformation rules to specific purposes of a particular project.

When specifying the rules for the transformation we assume that the requirements are specified in a precise manner using RSL as presented in the previous Section. In order to receive a four-tier architectural model we should apply the following set of general rules (refined from [106]; see also initial ideas formulated in [193]):

- Every vocabulary package is transformed into one business component and one data access component.
- Every notion used in an SVO[O] sentence is transformed into a data transfer object (DTO). Notions which occur only in "high-level requirements" and system vision are ignored. Every relationship between notions is transformed into an association between the corresponding DTO classes. Associations between DTO classes reflect the direction and multiplicities of the relations between appropriate notions (if exist).
- For every notion used in an SVO[O] sentence, a data access object (DAO) interface is generated in the corresponding data access component. This interface will provide CRUD operations for any given notion.

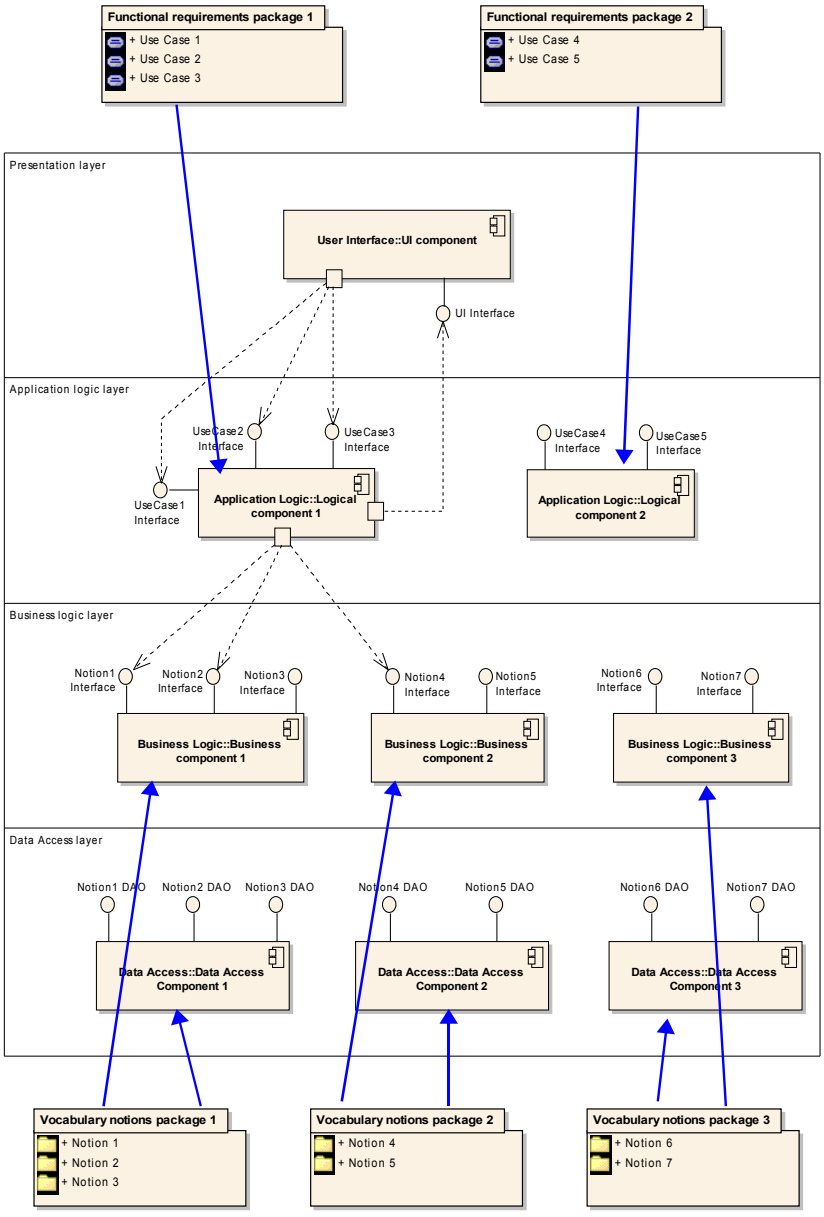

Fig. 4.12. Transformation of requirements into architecture - overview

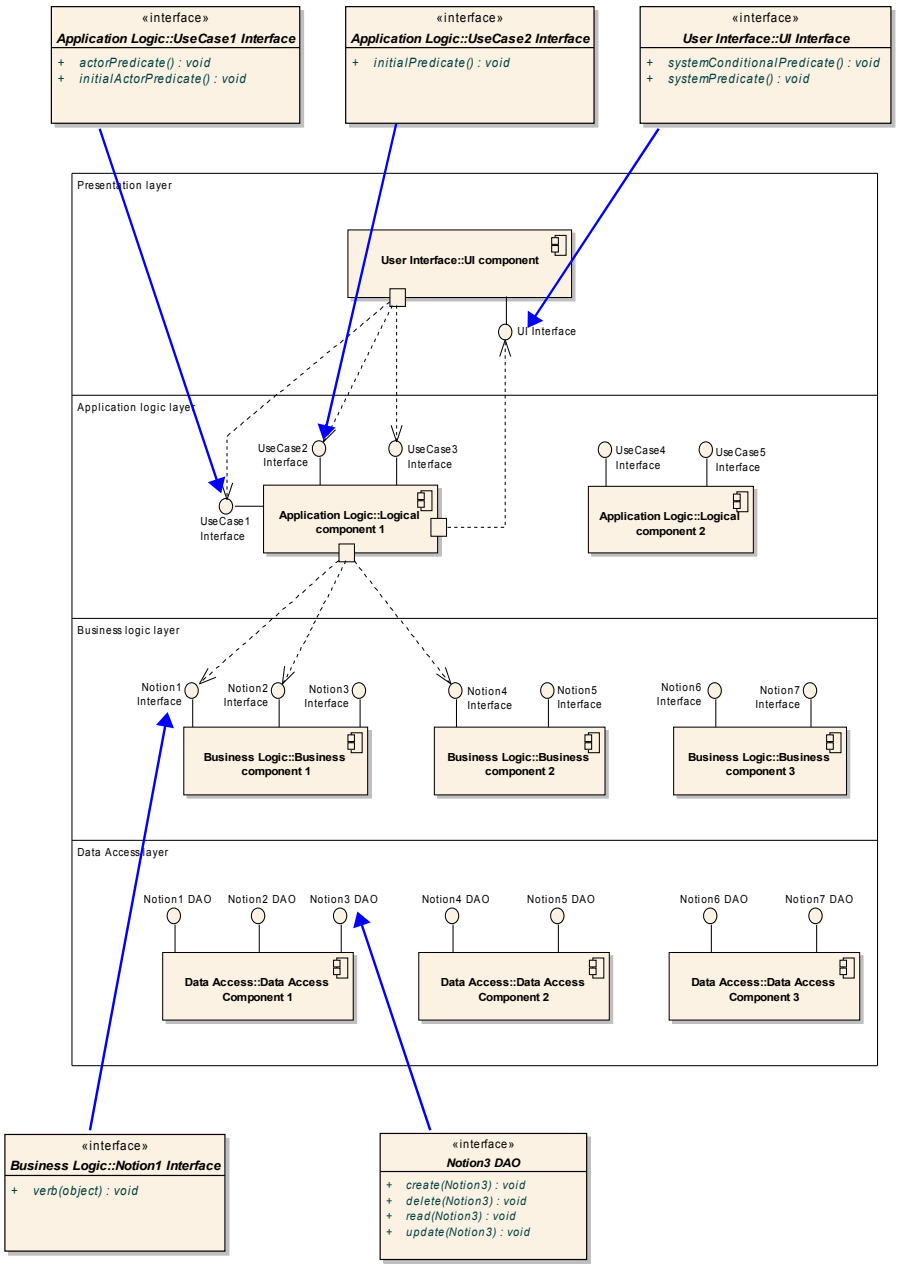

Fig. 4.13. Transformation of requirements into architecture - interfaces

- Every notion contained in a given vocabulary package is transformed into one interface provided by a business component. An interface corresponding to this vocabulary notion is generated only if the method for this notion is called on a business component (in the scenario we have a "self message" with this notion in the predicate).
- Every functional requirements package is transformed into one application logic component. It is important to keep the number of functional requirements in packages low (for instance: less than 4, specified by the transformation's parameter), to prevent creation of too complex components with too many interfaces (see also the following rule).
- Every use case (a functional requirement) is transformed into one interface provided by an appropriate application logic component.
- One UI component with one interface is generated for the whole application. Further division of this component can be done by an architect at a later stage.
- All actors are transferred with no changes.

These rules are illustrated in Figures 4.12 and 4.13. There, we can see two packages with functional requirements and three packages with notions. Every such package is transformed into one or two components in the appropriate tiers of the architectural model. Every use case and notion is transformed into an appropriate interface. For the notions, additional interfaces are generated in the Data Storage Tier. Figure 4.13 shows some details of the generated interfaces. The operations contained in them depend on more detailed rules that shall be presented below.

The above rules pertain only the static structure of the system built. We also need to assure that the system's functionality reflects the functional requirements. We thus need to transform use cases with their scenarios into architectural constructs. Here we shall use sequence diagrams as means to show the system dynamics. Messages in these diagrams shall be strictly associated with appropriate SVO[O] sentences in the original scenarios being the source of the transformation.

First we need to consider different types of SVO[O] sentences, and specifically their predicates which are to be used to name the target messages. The following types of predicates can be identified (see the previous Chapters and the previous Section for examples of SVO[O] sentences and their predicates):

- Initial actor predicate - part of the first sentence in a scenario, denoting actor's initial interaction with the system.
- Actor predicate - part of every further sentence denoting actor's interaction with the system (ie. every sentence with the actor being its subject).
- System response predicate - part of every sentence denoting system's activity in responding to the actor (ie. every sentence with the system being its subject, followed by a sentence with the actor being its subject).

- System self predicate - part of every sentence denoting internal system's activity (ie. every sentence with the system being its subject, followed by another such sentence)

Having the above terminology for predicates we can now present the following detailed transformation rules:

- Initial actor predicate (which represents an action initiating a UseCase) is transformed into two messages: first one from the actor to an appropriate presentation layer component and the second one from this presentation layer component to an appropriate application logic component's interface. The application logic component is the one that was transformed from the respective use case package according to the previous rules presented above. In addition, a dependency between the presentation layer component and the application logic component's interface used in this call should be created.
- Every actor predicate is also transformed using the above rule.
- Every system response predicate (description of an action that is a system's response to the actor's activity) is transformed into a message from the application logic component's interface to the presentation layer component's interface.
- Every «invoke» («include» or «extend», according to the UML specification of the use case model) construct is transformed into a message from the "current" application logic layer component (the one transformed from the package containing the currently transformed UseCase) to the application logic layer component's interface for the invoked UseCase.
- Every system self predicate is transformed into a message from the application logic layer component to a business layer component's interface. This business layer component corresponds to a notion which constitutes the sentence object in the current predicate. For this message, a dependency between the "source" application logic layer component and the "target" business layer component's interface used in this call should be created.
- For every of the above messages an operation is generated within a related interface.
- Messages to the business logic layer component's interface should correspond to verb phrases of the notion which corresponds to this interface:
  - If the verb phrase in the predicate is a simple verb phrase, then this verb is used for the operation's name, and the object part of the predicate is transformed into the operation's parameter. Example: [[v: add n: user]] => *add(User)*.
  - If the verb phrase is complex, then the verb and the direct object form the operation's name and both the direct and indirect objects are transformed into the operation's parameters. Example: [[v: add n: user p: to n: user list]] => *addUser(User, UserList)*.

In addition to the above, also some naming conventions should be set in the architectural model. Naming of elements in the requirements model is rather accommodated to the preferences of the users. The system implementors are used to different naming conventions and thus also transformation of names is needed. The following rules can be applied to rename elements that got transformed from notions, use cases, verb phrases, and so on:

- All element names should be converted to UpperCamelCase, e.g. *user list => UserList*
  Exceptions:
  (1) calls between actors and presentation layer components should remain in the same form as in SVO[O] sentences
  (2) operation names should follow the camelCase naming pattern (small letter in front).
- In the Business Logic tier all component names should have the word "Services" added at the end, e.g. *FacilityUses => FacilityUsesServices*.
- All interface names in all layers should have the letter "I" added in front, e.g. *FacilityDetails => IFacilityDetails*.
- All component names in the Data Access tier should be in plural form and have the words "DataAccess" added at the end, e.g. *Device => DevicesDataAccess*.
- All interface names in the Data Access tier should have the additional acronym "DAO" added at the end, e.g. *Account => IAccountDAO*.
- All notions (which form the data transfer objects) should have names ending with the acronym "DTO", e.g. *RejectionMessage => RejectionMessageDTO*.

Certainly, these naming rules should be flexible. The architects should be able to change the naming conventions according to the guidelines present in a given software development organisation or project.

The presented rules for generating messages and names are illustrated in Figures 4.14 and 4.15. Figure 4.14 illustrates transformation of different types of SVO[O] sentences (according to the classification of predicates given above) into messages in architectural sequence diagrams. The lower diagram in this Figure is equivalent to a use case scenario but transformed into a preliminary sequence diagram. Every message in this diagram is an exact equivalent of an SVO[O] sentence in a scenario written textually or with an activity diagram (see the previous Section). The upper diagram is transformed from the scenario according to the above given rules.

To make this illustration more specific let's take the example scenario given in subsection 4.1.2. The first sentence of this scenario is "Customer swipes n: card" which is the "initial actor predicate" sentence. The second sentence ("FC System verifies account balance") is a "system self predicate" sentence (marked simply as "self predicate" in Fig. 4.14) as the next one ("FC System opens utility gate") has also "system" as its subject. This next sentence is a "system response predicate" (marked simply as "system predicate" in the Figure). This

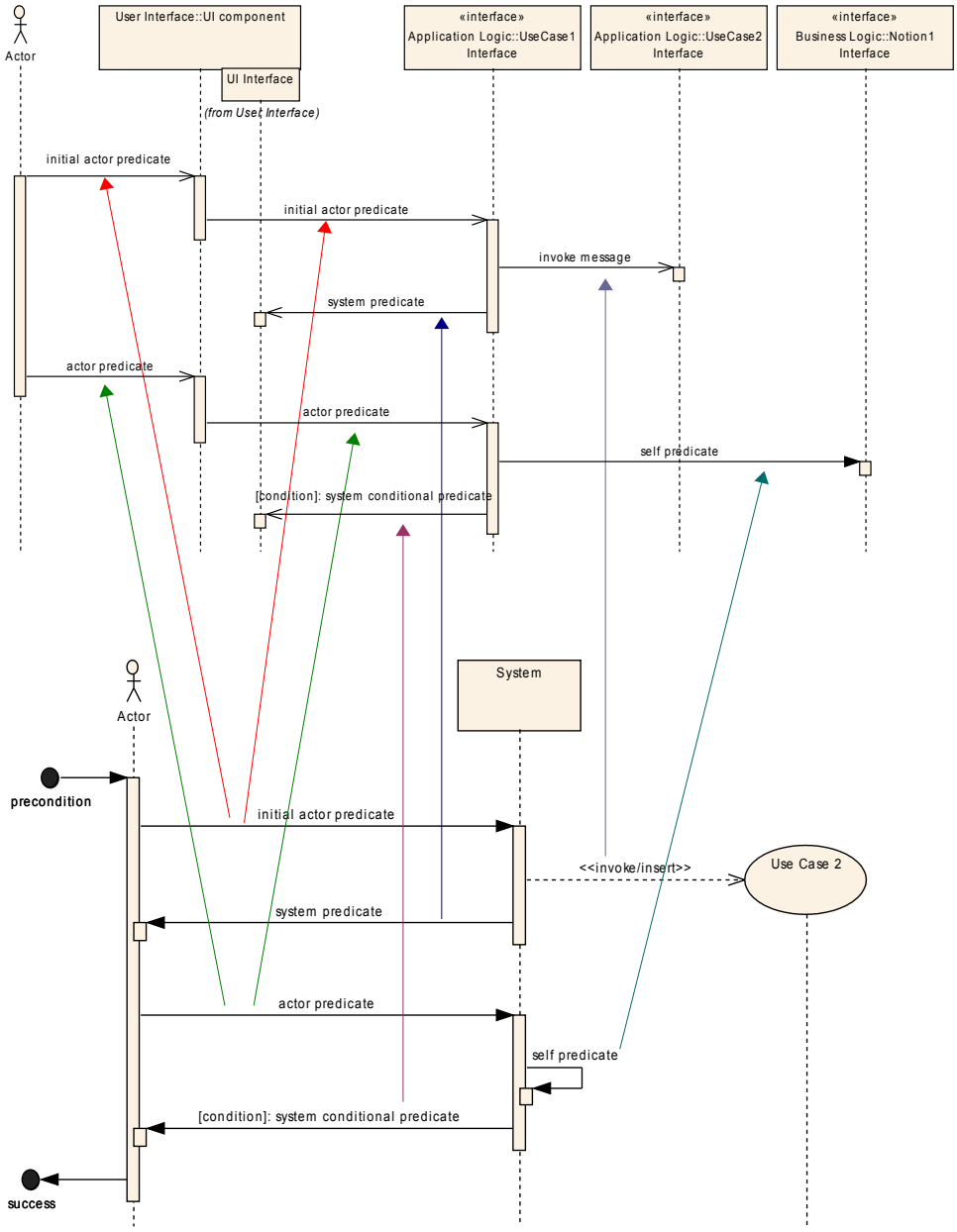

Fig. 4.14. Transformation of requirements into architecture - sequence diagrams

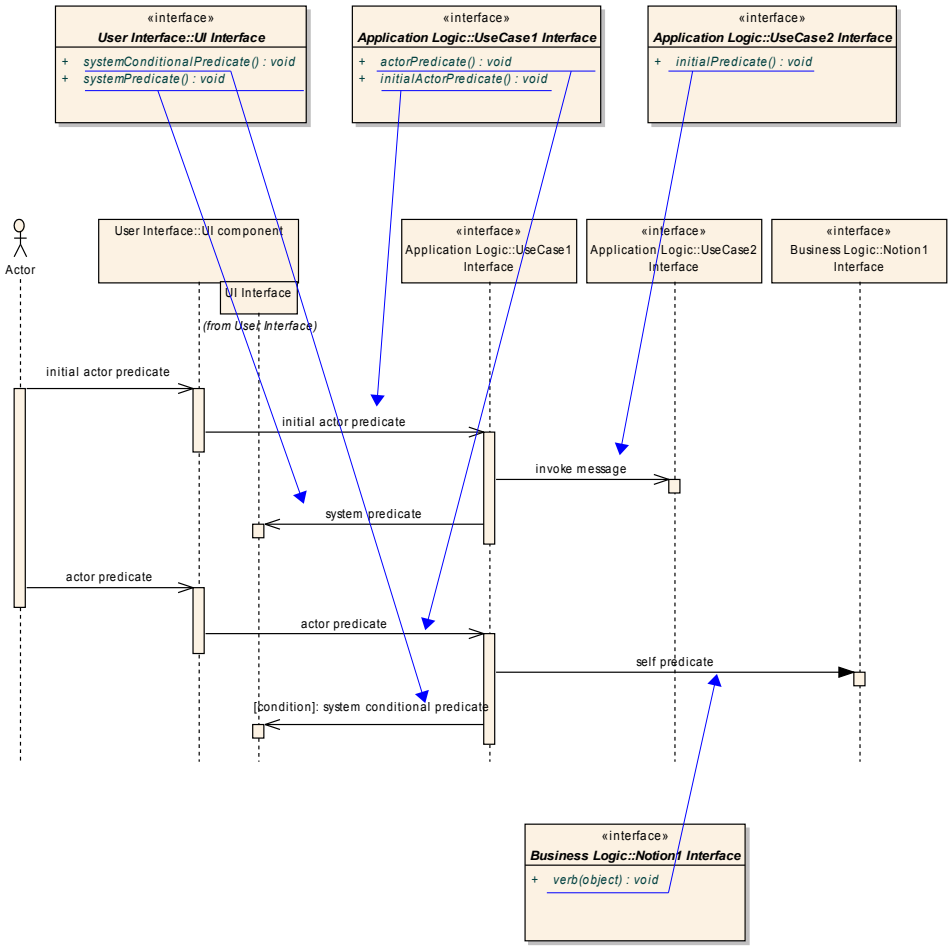

Fig. 4.15. Transformation of requirements into architecture - interface operations

is because of the following sentence ("Customer passes gate detector"), which is an "actor predicate". Finally we have two "system" sentences where the first one is a "system self predicate" and the last one is "system response predicate".

It can be noted that actually one of the "system" sentences is preceded by a condition in the scenario. This sentence is also preceded by this condition in the preliminary sequence diagram. Now, the arrows in Figure 4.14 show how to transform sentences (equivalent to messages in the preliminary sequence diagram) into messages in the architectural level sequence diagram. These arrows follow the rules presented above. Here we have a transformation which transforms a sequence diagram into another sequence diagram. Obviously,

a direct transformation from the equivalent purely textual scenario is also possible.

Figure 4.15 supplements the above example with the illustration of rules for generating interface operations. The arrows in this Figure show the relationships between message names and the names of appropriate operations, according to the presented transformation rules.

4.2.3 Rules for transforming architecture into detailed design

The architectural model generated with the rules presented in the previous subsection can be seen as independent of any specific implementation technology. We can call it the Platform Independent Model (PIM) according to the terminology used in MDA/MDD (see [138]). Now, the role of the architects is to choose a specific technology to be used for implementing the system. The architects thus should propose a specific platform in order for the designers to build the Platform Specific Model (PSM, again according to MDA/MDD).

Having the platform chosen we can define rules for transforming the PIM into PSM. This will give us the necessary detailed design model which can then be generated into code. Generally, the transformation consists in generating the contents of components in the four tiers of our architectural model. The transformation process uses only the information contained in the architectural model, assuming that transformation from requirements to architecture extracted all possible information for generating the detailed design model. Obviously, the chosen platform determines the rules of the transformation. Here we shall give an example of such rules for a generic implementation platform. It has to be stressed that the given set of rules is limited and far from being complete. Giving a complete set of rules is out of scope of this book as it would also necessitate presenting and giving an introduction to a specific technological platform.

For the **Application Logic Tier** we shall create a structure that enables the realisation of use case related operations of individual interfaces. The rules contain also certain naming patterns and relationships that may exist between elements in this tier.

- A single class serving as a general application logic factory is created. It is a static class with the name “AppLogicFactory”.
  - For every component in the application logic tier, an operation within “AppLogicFactory”, returning a component factory is generated. The method name is composed of a “get” prefix and the component name (e.g. getReservationsFactory)
- For every component in the application logic tier, a corresponding component factory is created. They are static classes with the name composed of the component name with a “Factory” suffix added (e.g. ReseervationsFactory)

  - For every interface in a given component, an operation returning a realisation of this interface is generated. The method name is composed of a "get" prefix and the interface name (e.g. getIBrowseAnOfferAndReserve)
- For every interface in the application logic layer, a corresponding interface and implementation class are generated (with realisation relationship).
  - Interface operations are exactly the same (copies) as in the architectural model.
  - The generated interface has exactly the same names as in the architectural model (e.g. IBrowseAnOfferAndReserve).
  - The generated implementation class has the same name as the interface without the "I" prefix (e.g. BrowseAnOfferAndReserve)
- For the application logic factory, dependencies to individual components' factories are generated.
- For every component factory, dependencies to implementation classes are generated. These dependencies point to each of the classes implementing the interface of this component.

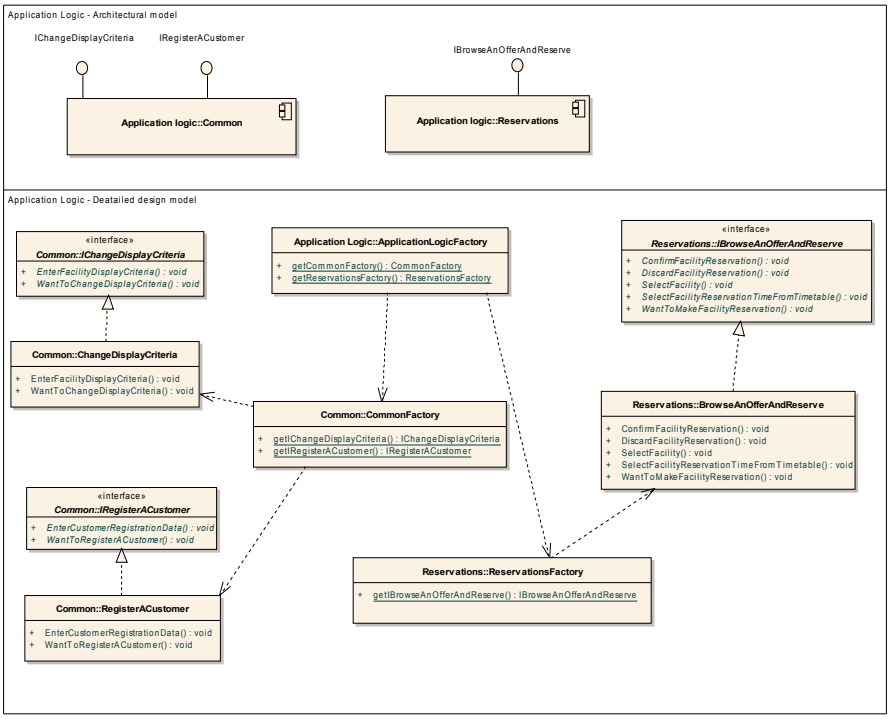

Fig. 4.16. Generation of application logic into detailed design

Figure 4.16 gives an example transformation according to these rules. Two application logic components are transformed into several classes connected with appropriate dependency and interface realisation relationships. Appropriate interface and class operations were also generated.

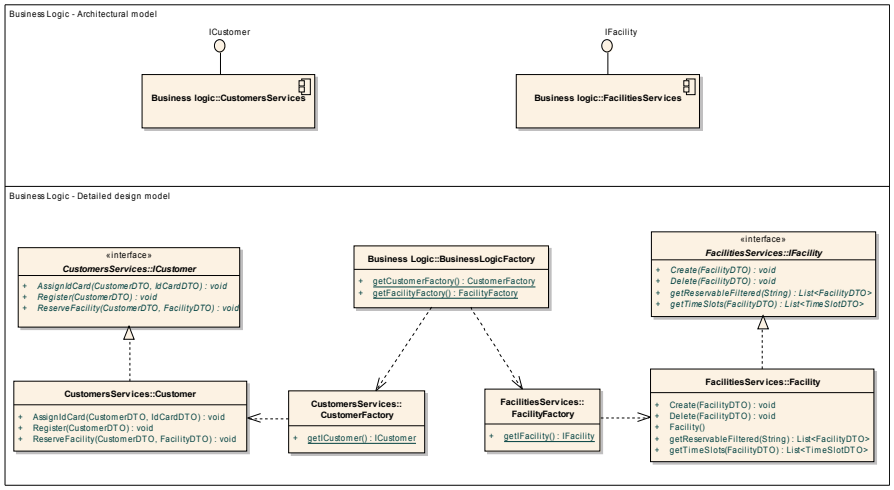

Fig. 4.17. Generation of business logic into detailed design

The **Business Logic Tier** is generated with a set of rules similar to those for the application logic. We shall omit these rules for brevity. The rules for the application logic exemplified in Figure 4.16 can be compared with an example for the business logic given in Figure 4.17. Similarly, the Data Storage layer can be generated.

As a final step of the transformation we need to generate relationships between the layers. These can be made on the basis of architectural sequence diagrams. Relationships between elements of the application logic layer and business logic layer can be generated by determining messages passed between these two layers. Messages between elements of the business logic layer and the data access layer are not generated during the transformation from requirements and thus should be set manually by the developers.

4.2.4 Model transformations as models

To perform a transformation compliant with the rules presented in the previous subsections we need to define it somehow. In fact, a language for defining transformations should be part of the overall case specification language as presented in Section 2.2.3. This language supplements the languages to define models themselves (like RSL or UML). It can be noted that the transforma-

tion specifications can be written visually and thus treated as models. Thus, the transformation language is in fact a modelling language.

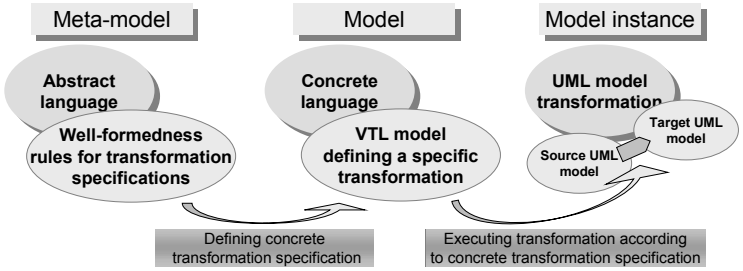

Fig. 4.18. Model transformation language architecture based on meta-modeling layers

The situation for the model transformation language is more complex than for a "normal" modelling language and thus needs some explanation. The concept is illustrated in Figure 4.18 (see also [198]). We use MOF to define the meta-model of the language. This is similar to the approach for defining RSL (see the previous Section) or UML. The specification of the language is on the "meta" level where the abstract language is defined. Specific models in the transformation language are specified with a concrete language (with a concrete notation). This gives us transformation specifications that can be executed. It is important to note that the execution of a transformation consists in taking a source model (conforming to a specific meta-model; of RSL for instance) and generating the target model (conforming to a perhaps another meta-model; of UML for instance). This results not only in the target model but also in a set o mappings between the source and the target.

Here we shall describe the meta-model of a simple transformation language. This definition is far from being complete, as defining a complete language is certainly out of scope of this book. Interested readers can be referred to definitions of languages like QVT (see [166]) or MOLA (see eg. [105]). Despite not being complete, the current description should supply the reader with an idea on how to design such languages and situates it within the overall software case meta-model (see Figures 2.10 and 2.6).

In this subsection we shall present the meta-model of the proposed transformation language which we shall call the Visual Transformation Language (VTL). In the next subsection, the concrete syntax and a simple transformation example shall be given.

The source and target elements of our transformations represent appropriate elements found in RSL or UML. We normally want to transform classes, interfaces, use cases, actors, components or SVO[O] sentences. Such elements

can be connected through various relationships: associations, dependencies or generalisations. In order to perform a transformation on such models we have to find an appropriate pattern, composed of source elements and relationships between them. Having such a pattern we can apply certain transformation rules. These rules determine the actual mapping from source elements and relationships into target elements and relationships. Our transformation language should, thus, be capable of defining templates for patterns in the source model, rules that describe mapping of elements consistent with these templates, and templates for relationships in the target model.

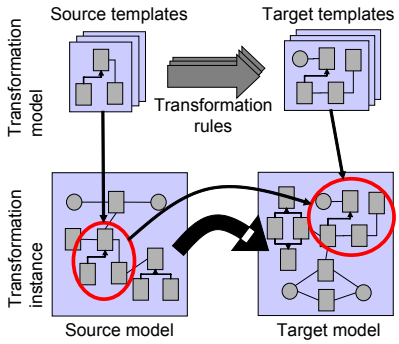

Fig. 4.19. Transformation rules and their application

The above requirements for our language are illustrated in Figure 4.19. Every transformation specification should contain certain "source templates" and "target templates", these being supplemented by "transformation rules". The source templates are sought in the source model. Then, a target model is generated according to the target templates and transformation rules.

These rules are reflected in the meta-model of our transformation language, shown in Figure 4.20 (this can be compared with Fig. 2.10). Every Transformation is composed of several source and target ComplexElementTemplates and several ComplexTransformationRules. It can be noted that all these elements of our metamodel are derived from the Package meta-class found in the UML specification (in the Kernel package, see [151]). Basic elements of every Transformation are TransformationElements that reflect elements in the source and target models of a transformation instance. These TransformationElements participate in SimpleTransformationRules and SimpleElementTemplates. Every simple rule is composed of an appropriate link (TransformationLink or TemplateLink) that connects two TransformationElements.

Figure 4.21 shows important details of individual elements in our metamodel. Every TransformationElement has its type. This type is defined with

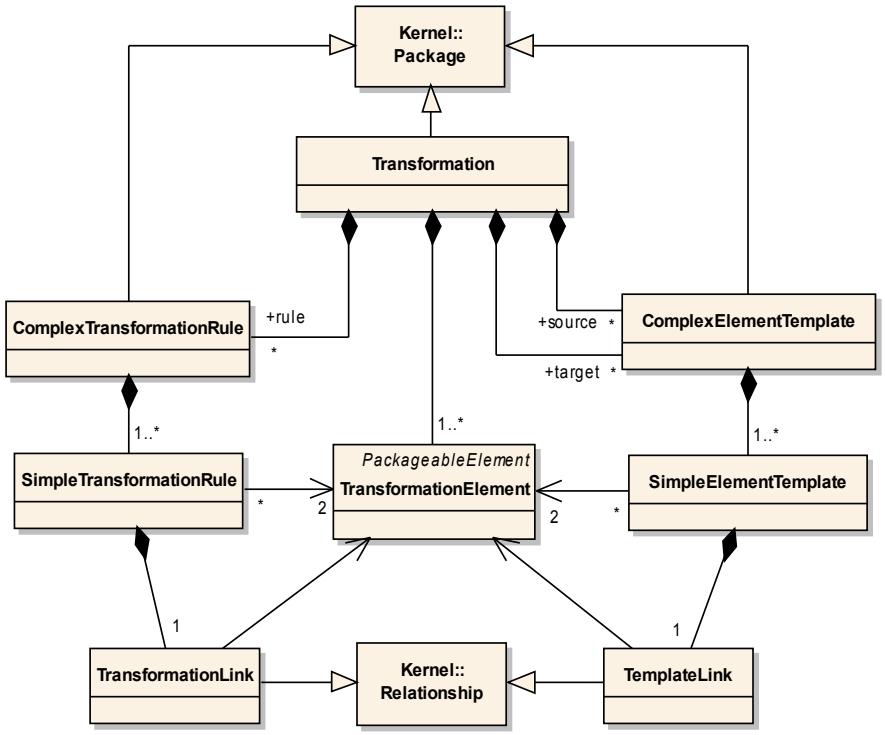

Fig. 4.20. Meta-model of the VTL transformation

the ElementTypes enumeration, which can have values reflecting various meta-classes found in the UML metamodel (like: Class, UseCase, Inteface an so on). The transformationSide can be either source or target. This determines, whether the element can participate in source templates or in target templates. An important attribute of the TransformationElement meta-class is the stereotypeName. The stereotype allows for performing more fine-grained transformations than only based on element types. By setting the value of this attribute in a transformation element, the transformation developer can determine differences in transforming model elements with different stereotypes. Another important meta-attribute of the TransformationElement is the packageName. The value of this attribute specifies the package in the source model where the element should be sought for or the package in the target model where the element should be placed in.

The remaining two meta-attributes of TransformationElement define the type of name conversion and member conversion respectively. Name conversion is very important, as it determines how the name of the current element will be changed in the element on the other side of an appropriate Transfor-

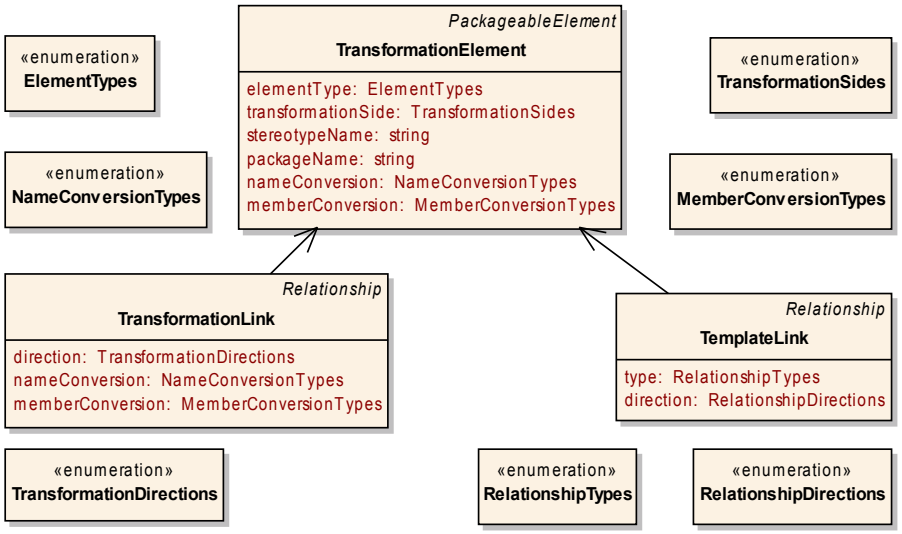

Fig. 4.21. Definition of the VTL meta-model elements

mationLink. One possibility is just to copy the name. Another two possibilities include adding a prefix or a postfix. Member conversion for a transformation element specifies how the elements members (like attributes, parts or operations) will be converted in the other element.

Name and member conversions are performed on elements (only between a source and a target element) connected with TransformationLinks. It can be noted that these links are directed. The direction determines whether this particular transformation can be performed between source and target only or in both directions. We can also observe, that the TransformationLink has the same conversion attributes as TransformationElement. This is due to the fact that a single element can be converted to several other elements. In such situation each TransformationLink coming from a source element can add its own control over conversion in addition to standard conversion specified inside this element.

For the definition of transformation to be complete, we also need to define the TemplateLink. This connector determines relationships between elements in the source or target templates. TemplateLinks can have types that reflect meta-classes of the UML specification that derive from the Relationship meta-class. Thus, we can have TemplateLinks typed eg. as Dependencies, Associations or Generalisations. TemplateLinks can connect only two TransformationElements on the same transformationSide. This allows for defining templates for the source or target models.

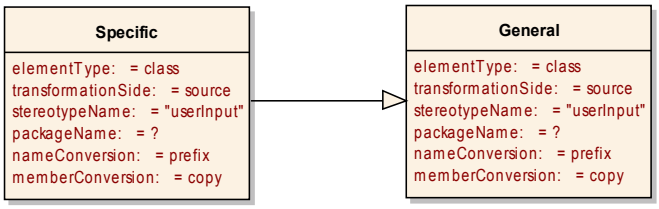

Fig. 4.22. Example source template

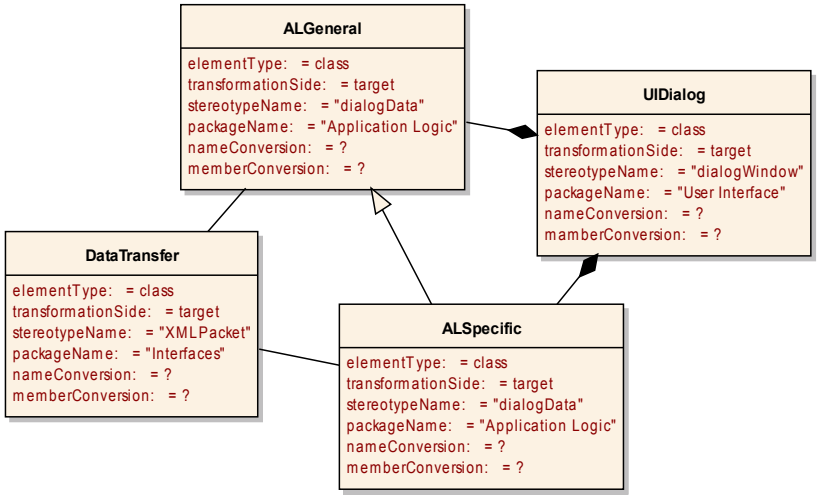

Fig. 4.23. Example target template

4.2.5 Specifying transformations

In order to specify a transformation we shall update the meta-model with concrete syntax. This is illustrated in Figures 4.22-4.24. These diagrams contain a VTL model for a specific transformation. As it can be noted, VTL TransformationElements are denoted with a symbol identical to the UML's class symbol. Also attributes and links are similar to those in UML class diagrams.

Figure 4.22 shows the source template of our transformation model. As we can see, the transformation is performed, whenever a generalisation relationship between two classes is found. Moreover, these two classes should be stereotyped as «userInput». The transformation is independent of the package in which the template elements are found (packageName = ?). Default name conversion for both of the classes is to add a prefix.

Source model fragments that match template shown on Figure 4.22 will get transformed into model fragments matching template on Figure 4.23. This

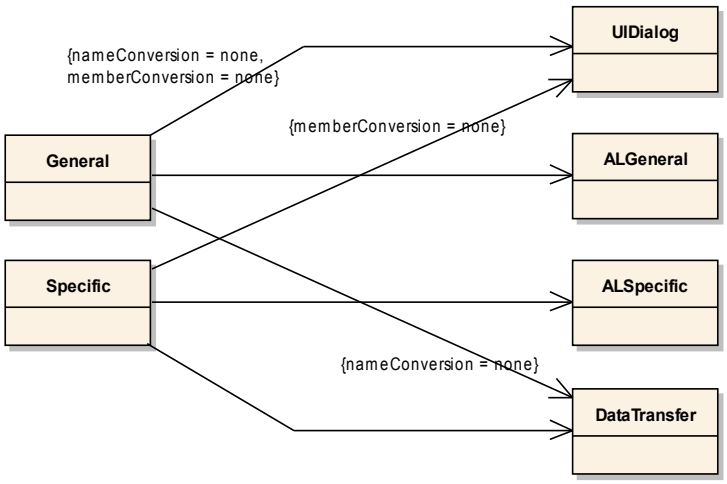

Fig. 4.24. Example transformation rules

target template contains four TransformationElements representing four classes in the target model. After transformation, these classes will be related with a generalisation, two aggregations and two associations. Their stereotypes will be set - according to the template - to «dialogData», «XMLPacket» and «dialogWindow». These classes will be placed in appropriate packages (User Interface, Application Logic and Interfaces). It can be noted that name and member conversion is not defined, which suggests that the transformation is uni-directional (only from source to target).

The transformation model is completed with transformation rules, shown in Figure 4.24. These rules determine which source elements will get converted into which target elements. The arrows show direction of this conversion (from source to target). Some TransformationLinks are adorned with constraints. These constraints override standard conversion rules defined in appropriate source TransformationElements. As we can see, the UIDialog target elements will have names converted from the Specific source elements only (with a prefix, according to Figure 4.22). No members will be copied into UIDialog elements. On the other hand, the DataTransfer elements will have member lists being a concatenation of members from General and Specific elements.

The above described transformation model is the basis for performing the actual transformation on UML models. An example of such a model is shown on Figure 4.25. This model contains three classes stereotyped exactly as specified in the source transformation template. In this model we have two fragments that match correctly the template from Figure 4.22. It can be noted that these two fragments have a common General class - the UserData class.

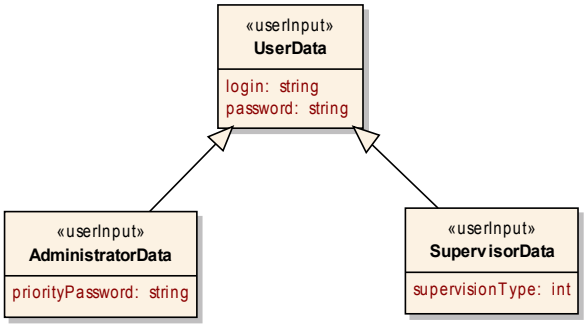

Fig. 4.25. Example source model ready to be transformed into a target model

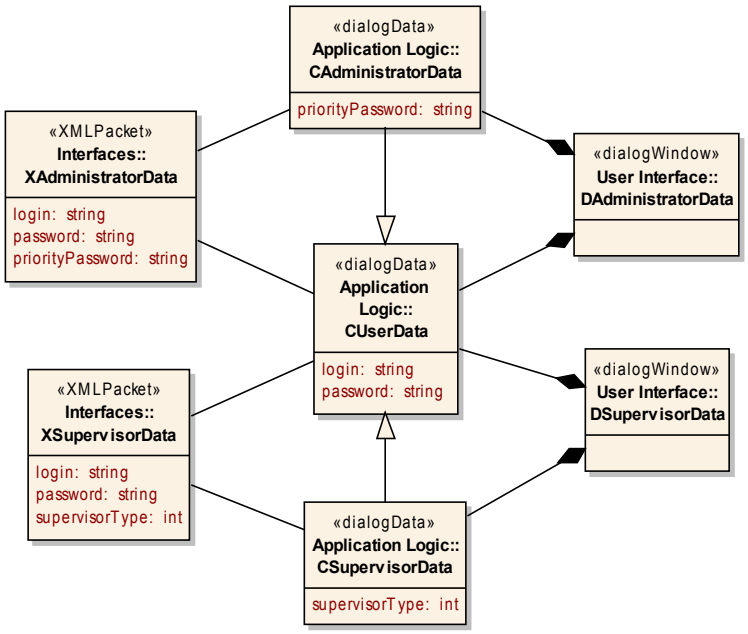

Fig. 4.26. Example target model transformed from the source model according to specified rules

After performing a transformation, the source model gets transformed into the model shown in Figure 4.26. This diagram contains two fragments that are derived from the template in Figure 4.23. These two fragments are again joined by one of the classes (the CUserData class). The joining class is transformed from the joining class in the source model. It can be noted, that the class name has been extended with a prefix (letter 'C'). Both attributes of UserData

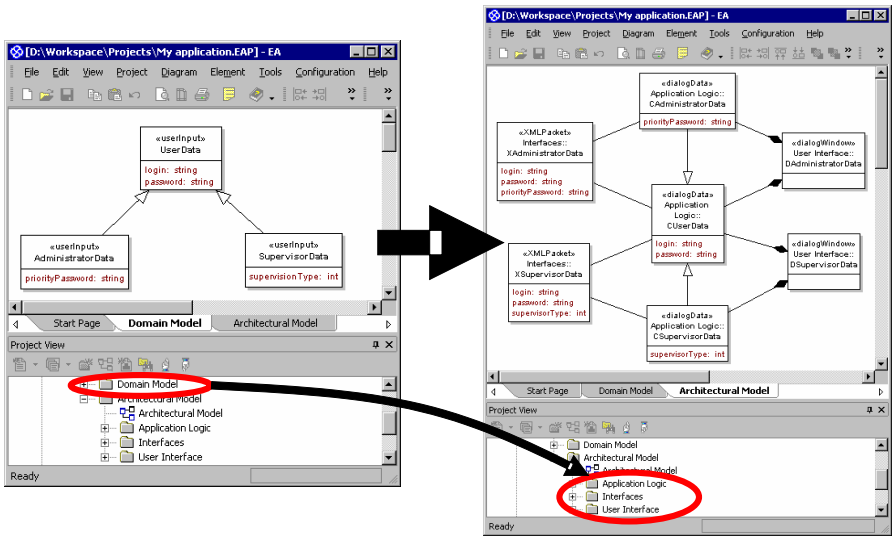

Fig. 4.27. Transformation in a CASE tool

have been copied into CUserData. Attributes from UserData have also been copied into XADministratorData and XSupervisorData. These two classes have also attributes copied from two other classes from the source model. This is consistent with the transformation rules shown on Figure 4.24.

The above example shows a very simple transformation of a requirements domain model into a design model. For more complex transformations, reflecting all the rules presented in the previous subsections we would need a much more complex meta-model for our Visual Transformation Language. However, even with this simple example it can be seen that the vision of supporting developers by generating architectural and detailed design models is feasible in practice. In Figure 4.27 we can see an example tool with the transformations applied. This illustrates the way software developers could use the transformation specifications in their everyday work. This is also a realisation of the ReDSeeDS Engine use cases, as presented in subsection 2.3.1.

4.3 Reuse mechanisms based on software cases

The previous two Sections of this Chapter present the details of specifying coherent software ceses. Having such means to define software cases, we also would like to be able to reuse them. This would fulfil the vision described in Chapter 2. Here we shall propose a simple mechanism for finding similar cases and retrieving them from a software case repository.

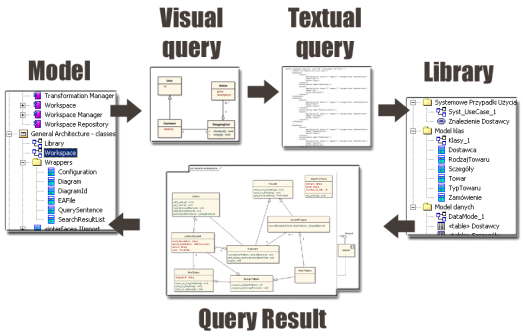

Fig. 4.28. Applying a reusable software case library

4.3.1 Reusable software case repository concept

The general concept of the software case repository is presented in Figure 4.28 (see [9] and [8] for more details). The library system is associated with two models. The first of the models contains the initial sketch of the problem or its solution - this is the developer's current workspace. The second model is composed of many individual, searchable software cases related to various problem domains, and forms the actual repository. The first model is the source of visual queries, usually equivalent to single RSL or UML diagrams (with several domain elements, use cases, components, scenarios etc.). These visual queries are transformed into text which can be parsed and used as the actual query for the pattern library. The software cases in the library can also be transformed into a textual representation which can be compared to the query. After performing appropriate comparison, the library's search engine returns several results based on their relevancy for the query. These results can then be incorporated into the developer's workspace and integrated with the rest of the model. It can be noted that the described approach is similar to the idea of applying reuse through ontology matching [16], here equivalent to matching RSL or UML models. It also needs to be stressed that the query language presented here serves the purpose of finding similar diagrams (or: graphs). This is significantly different to an object database query language which serves the purpose of finding data in a relational or object oriented structure (see the descriptions of OQL [33] and SBQL [202]).

4.3.2 Query meta-model

The query meta-model is based on an assumption that every two related elements from the pattern library can be represented as a triple: object-relation-object. Such a triple denotes a single connection between the source model elements. This connection might represent eg. associations or dependencies

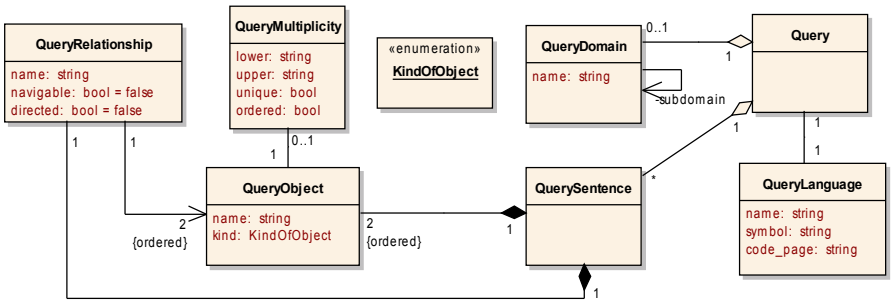

Fig. 4.29. General query meta-model

between model elements or membership of element features (like domain element attributes or phrases).

According to Figure 4.29, a query is composed of several sentences. Every such QuerySentence contains the above described triple: two ordered QueryObjects and a QueryRelationship that connects these two objects. Each of the objects can be adorned with appropriate QueryMultiplicity. The query as a whole can be associated with (narrowed to) a specific QueryDomain. We can also specify the QueryLanguage which allows for applying queries in different natural languages (with an appropriate language-to-language dictionary).

Objects in a query sentence can represent various meta-classes found in the UML or RSL specifications. This is reflected by the kind attribute of the QueryObject, that can have values dependent on the KindOfObject «enumeration». These values include (but are not limited to) the following types of elements : classes, interfaces, components, attributes, operations, parameters, actors, use cases. Figure 4.30 shows general mapping of these various types of model Elements onto QueryObjects. The value of the kind attribute depends on the actual meta-class derived from the Element which is taking part in building a query. It can be noted that QueryRelationship has no specified type. Instead, the name attribute is used to distinguish eg. between an aggregation and an association. The other features of Relationships reflected in the queries are their direction and navigability. It can be noted that for an RSL domain model this direction is irrelevant, however the presented language can also allow for comparing design models.

Figure 4.30 does not show how membership relationships are mapped onto QueryRelationships, as we do not have a separate "membership" meta-class in UML or RSL. If an Element is a member of another Element this gets reflected in creating a QuerySentence with the two elements mapped onto two QueryObjects and a relationship named "Ownership".

Another important element of a query is the multiplicity of participating objects. QueryMultiplicity can be attached to a QueryObject if the mapped

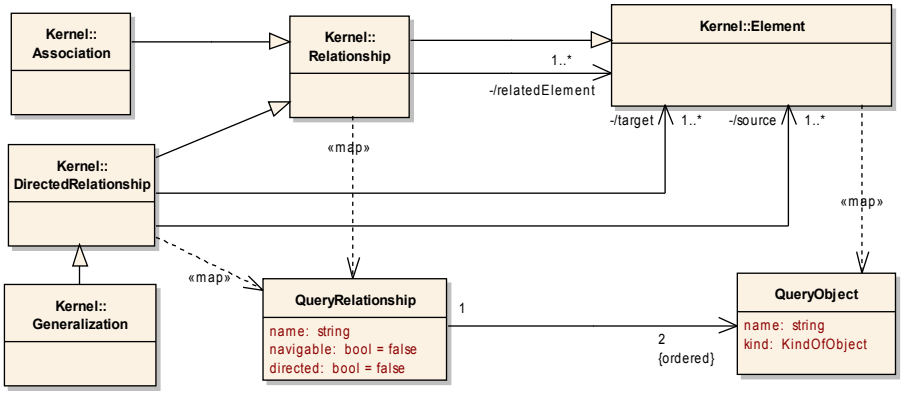

Fig. 4.30. General mapping of model elements and relationships onto query elements

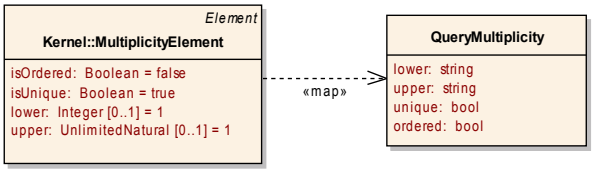

Fig. 4.31. Mapping of multiplicities

model element is adorned with multiplicity. The actual mapping is shown in Figure 4.31.

By applying a query, based on the above described meta-model, we can compare various model elements, like classes, interfaces, components, operations, parameters, attributes, actors, use cases, use case scenarios, scenario sentences or sentence parts (sentence subjects, verbs or objects). The mapping depends on the value of the kind attribute of the QueryObject. Similarly, the value of the kind attribute of QueryRelationship determines the type of relationship between queried model elements. Appropriate mapping between query sentence parts and the most general UML meta-model elements with relationships is presented in Figure 4.30. This mapping is quite obvious and needs no further explanation. Similarly, there is no need to explain the mapping of the query's multiplicity, shown on Figure 4.31.

In addition to the above presented general mappings, Figures 4.32-4.35 show more specific mappings for the class model (or RSL domain model) and the use case scenario model. These mappings are most important for the types of queries needed to fulfill requirements level reuse.

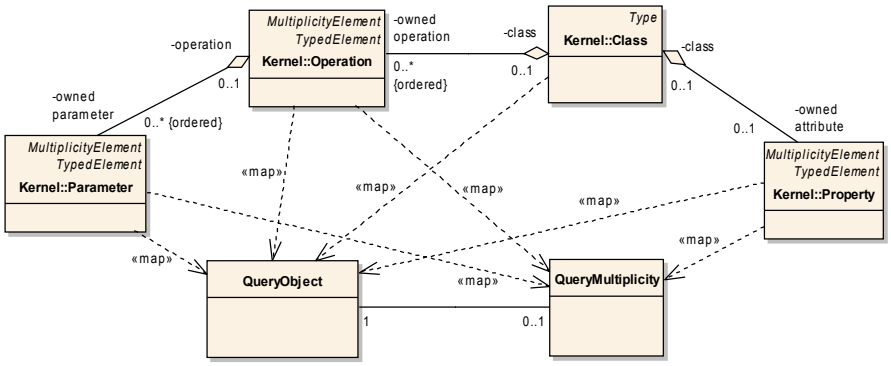

Fig. 4.32. Mapping of classes and domain elements

When mapping a class model we need to transform classes, associations and dependencies between them, with their properties and operations. This transformation mapping is shown in Figure 4.32. For every single class there can be built several QuerySentences. These sentences can describe relations between this class and another related class or between the class and its attributes or operations. One of the QueryObjects in a sentence is mapped to this specific class, and another object is mapped to another class or one of the class features. Also, operation parameters can form QueryObjects in relation to appropriate operations in a query sentence. In a class-related query, an important element is the multiplicity of elements in the mapped relationship. This might reflect the multiplicity of a role in an association or the multiplicity of an attribute. Figure 4.32 does not show mapping of relationships and memberships, as this is consistent with the general mapping described above. It can be noted that this mapping can be performed in an analogous manner for the RSL domain model which is similar to the class model described here as being more general.

Mapping between use cases should be performed on several levels. On the highest level, QueryObjects are mapped to UseCases and Actors (see Fig. 4.33). Relationships between use cases are also mapped appropriately. What is more important, we can also map the structure of individual use case scenarios. We can assume that an individual UseCase has several associated Scenarios with ordered ScenarioSentences. Mapping of use case structural elements into query elements is shown in Figures 4.34 and 4.35. Scenarios, sentences and sentence elements are mapped into appropriate QueryObjects. QueryRelationships reflect aggregation between scenarios, their sentences and sentence parts.

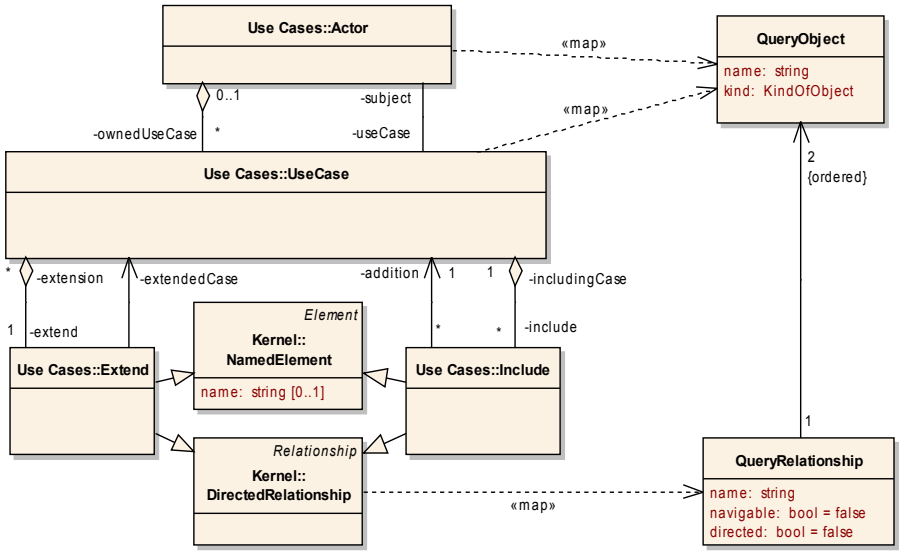

Fig. 4.33. Mapping of use cases

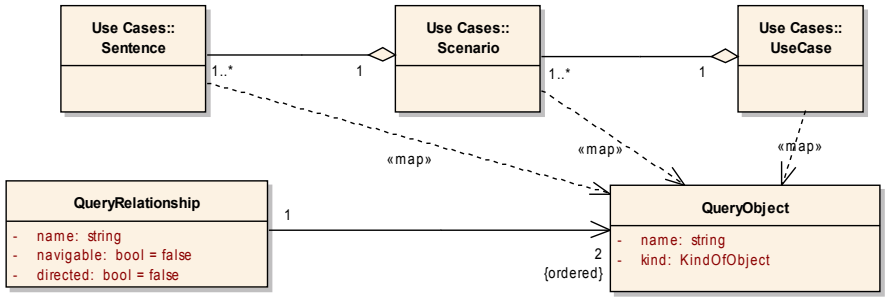

Fig. 4.34. Mapping of use case scenarios

4.3.3 Creating queries

Sentences consistent with the abstract syntax defined in the previous subsection can be represented in textual form in the following simple concrete syntax for a single query sentence:

```
[object multiplicity] [object type] [object name] [relationship] [object multiplicity] [object type] [object name]
```

Which is illustrated in the following example:

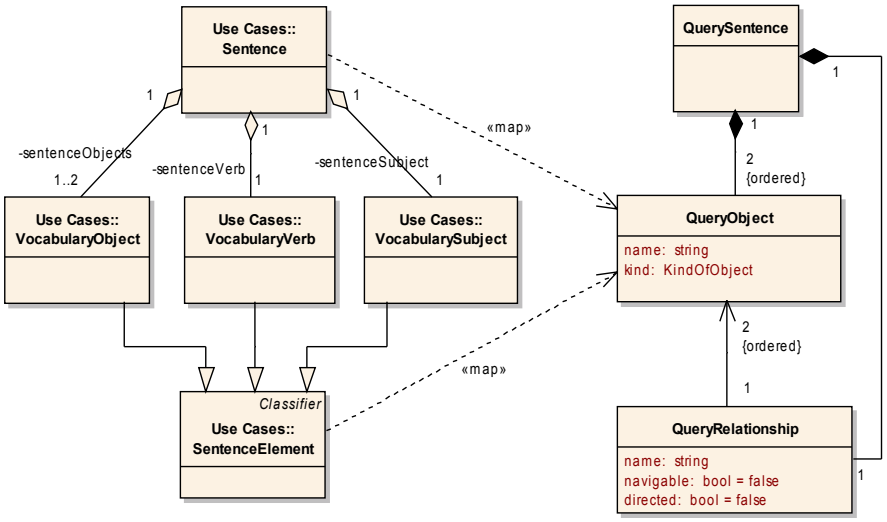

Fig. 4.35. Mapping for a single scenario sentence

```
[0,1] [class] [ClassName] [aggregation] [?,?] [class] [ClassName2]
```

This sentence denotes a relationship between two classes in an aggregation relationship. The multiplicity of one of the classes (more correctly: classe's role) in this relationship is unspecified, and another class has multiplicity of [0..1]. Having defined the query metamodel in section 3 we can now easily write the structure of this query in XML:

```
<query language="English" code="EN" code_page="ISO-8859-1">
  <domain="domain name/subdomain name"/> <sentence>
    <object>
        <multiplicity lower="0" upper="1" unique=true ordered=true>
        <name="ClassName">
        <kind="Class">
    </object>
    <relationship name="Aggregation"/>
    <object>
        <multiplicity lower="?" upper="?" unique=? ordered=?>
        <name="ClassName2">
        <kind="Class">
    </object>
</sentence> </query>
```

To illustrate application of queries for the purpose of requirements matching we will now build two more complex queries. The first query will be built by transforming it from a class diagram shown in Figure 4.36. Treating this

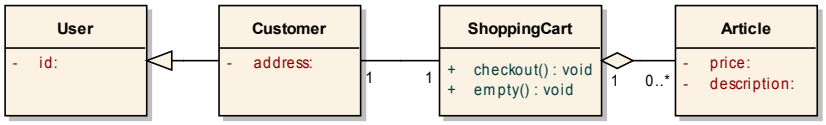

Fig. 4.36. Class diagram with an example query

diagram as a visual input we can automatically build a query, which is shown (in a fragment) below:

```
<query language="English" code="EN" code_page="ISO-8859-1">
<domain="E-commerce/On-line shop"/> <sentence>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered=true/>
        <name="User">
        <kind="Class">
    </object>
    <relationship name="Ownership"/>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered=true/>
        <name="id">
        <kind="Attribute">
    </object>
</sentence> <sentence>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered=false/>
        <name="User">
        <kind="Class">
    </object>
    <relationship name="Generalization"/>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered=true/>
        <name="Customer">
        <kind="Class">
    </object>
</sentence> (some other sentences...) <sentence>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered=false/>
        <name="ShoppingCart">
        <kind="Class">
    </object>
    <relationship name="Ownership"/>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered=true/ >
        <name="checkout">
        <kind="Operation">
    </object>
```

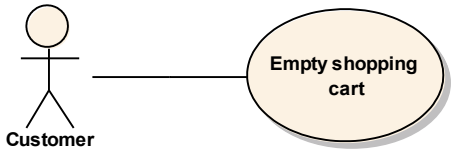

Fig. 4.37. Use case diagram with an example query

```
</sentence> (some other sentences...) <sentence>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered=false/>
        <name="ShoppingCart">
        <kind="Class">
    </object>
    <relationship name="Aggregation"/>
    <object>
        <multiplicity lower="0" upper="*" unique=false ordered=false/ >
        <name="Article">
        <kind="Class">
    </object>
</sentence> </query>
```

The second example shows a query for the use case model shown in Figure 4.37. In addition, the single use case in the model has the following scenario:

1. User presses Empty Shopping Cart button
2. System empties the shopping cart.

We can note that the single UseCase from our example has an associated single Scenario with two Sentences. Each of the two sentences is composed of three VocabularyElements (a sentence subject, a sentence verb and a sentence object). Sentences are numbered and the number constitutes that sentence's name. Considering the above relationships we can generate a query, where its fragment is shown below:

```
<query language="English" code="EN" code_page="ISO-8859-1">
<domain="E-commerce/On-line shop"/> <sentence>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered=false/>
        <name="Customer">
        <kind="Actor">
    </object>
    <relationship name="Ownership"/>
    <object>
        <multiplicity lower="1" upper="1" unique=true ordered= false />
        <name="Empty Shopping Cart">
```

```
            <kind="Use Case">
        </object>
    </sentence> <sentence>
        <object>
            <multiplicity lower="1" upper="1" unique=true ordered=false/>
            <name=" Empty Shopping Cart">
            <kind=" Use Case">
        </object>
        <relationship name="Ownership"/>
        <object>
            <multiplicity lower="1" upper="1" unique=true ordered= false />
            <name="Main path">
            <kind="Scenario">
        </object>
    </sentence> <sentence>
        <object>
            <multiplicity lower="1" upper="1" unique=true ordered=false/>
            <name="Main path">
            <kind="Scenario">
        </object>
        <relationship name="Ownership"/>
        <object>
            <multiplicity lower="1" upper="1" unique=true ordered=false/ >
            <name="1">
            <kind="Sentence">
        </object>
    </sentence> <sentence>
        <object>
            <multiplicity lower="1" upper="1" unique=true ordered=false/>
            <name="1">
            <kind="Sentence">
        </object>
        <relationship name="Ownership"/>
        <object>
            <multiplicity lower="1" upper="1" unique=false ordered=false/ >
            <name="user">
            <kind="Subject">
        </object>
    </sentence> <sentence>
        <object>
            <multiplicity lower="1" upper="1" unique=true ordered=false/>
            <name="1">
            <kind="Sentence">
        </object>
        <relationship name="Ownership"/>
        <object>
            <multiplicity lower="1" upper="1" unique=false ordered=false/ >
            <name="presses">
            <kind="Verb">
```

```
    </object>
</sentence> </query>
```

Queries like the ones presented above can be generated automatically (through defined mappings) from the visual model or created "by hand" by the developer. It is also possible that the initially generated query is modified or supplemented with additional query sentences.

The algorithm for automatic query generation is as follows:

1. for a given input model package gather all the packaged elements from the model repository;
2. for every gathered element:
   a) determine its type and name;
   b) for all the relationships of the element determine:
      - source and target element;
      - multiplicity of the source and target;
      - type of relationship;
3. form QuerySentences out of QueryObjects, QueryRelationships, and QueryMultiplicities created in step 2
4. form a Query out of created QuerySentences

Such an automatically generated query can now be supplemented or edited by the developer. The final query can be transformed into an XML packet and sent to the search engine in order to be applied to the pattern library.

4.3.4 Applying queries

The number of query sentences in a query is not limited. It depends on the complexity of the source model and the number of sentences added to the query by the library user. For models containing only a few classes or use cases with several attributes, operations or scenarios, the number of query sentences grows into tens, and even hundreds. Having such large queries, an effective way of comparing the query with the stored software cases has to be applied.

For every software case inserted into the library, an appropriate XML-based textual description is generated. These descriptions form an index that significantly accelerates query application. Every query applied is now compared with these index entries in order to calculate relevance points for each of the patterns.

The process of applying a query to a pattern library is illustrated in Figure 4.38. Having the initial model (query pattern), an appropriate query is generated. Query sentences in the generated list can be browsed and modified. The final query is then applied and several software cases are found. For each of the software cases found, a number, denoting its relevance (similarity with the initial pattern) is shown.

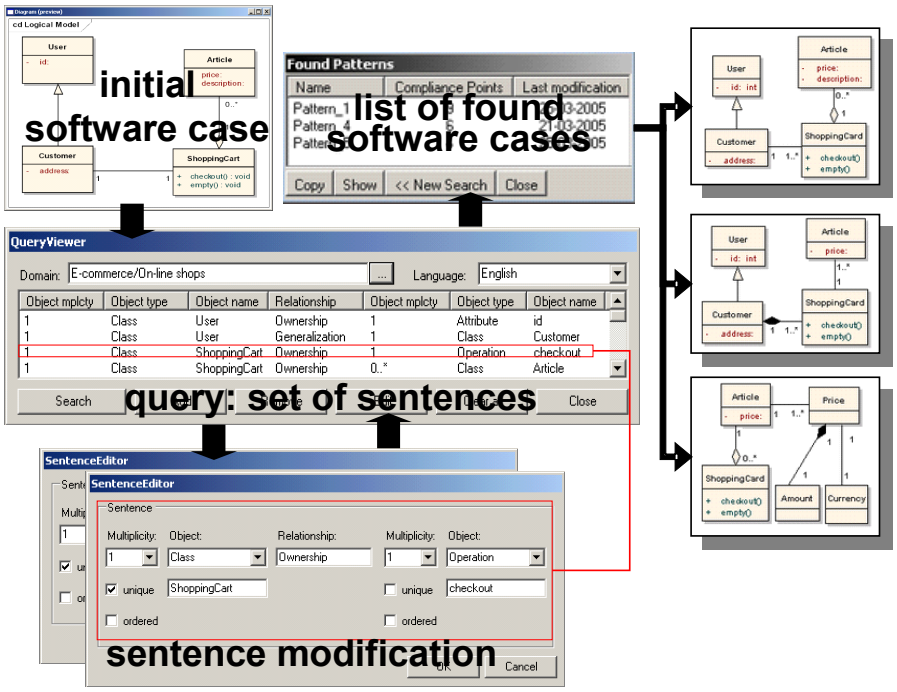

Fig. 4.38. Illustration of the search process

The relevance number reflects the number of query sentences that are matched with sentences of index description for a given library pattern. Sentences are compared by matching their objects and relationships as shown in Figure 4.39. When calculating relevance numbers, appropriate weights can be applied. These weights may be assigned to different object and relationship types and are set by the users before applying a query.

After obtaining a list of relevant software cases, the user can browse them and choose those that are suited best for the current problem. These chosen cases can be incorporated into the current workspace and included in the final solution.

In Figure 4.38 we can see that three relevant software cases have been found in the library. The most relevant case has 9 relevance points. It is in fact identical with the initial query (see also Fig. 4.36), and thus, from 9 compared sentences, 9 were matched correctly:

- Class User owns attribute id
- Class Customer owns attribute address
- Class Customer is derived from class User
- Class ShoppingCart own operation checkout()

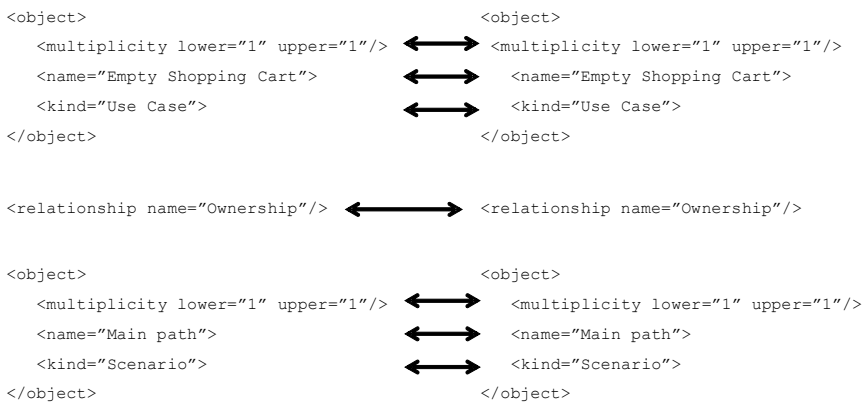

Fig. 4.39. Calculation of relevance value for a single query sentence

- Class ShoppingCart own operation empty()
- Class Customer is associated with class ShoppingCart with multiplicities [1..1] and [1..1]
- Class Article owns attribute price
- Class Article owns attribute description
- Class ShoppingCart aggregates class Article with multiplicities [1..1] and [0..*]

Obviously, the final query scheme should be much more sophisticated in order to achieve the software reuse vision formulated throughout this book. However, even with such simple queries, relevant software cases can be found. By querying requirements models only, we obtain complete solutions, with mapped architectural models, design and code. What is also important, we can also retrieve transformation definitions. With these, a new, adapted solution can be generated from the current requirements model, according to the description earlier in this Chapter.

5

Summary and discussion

In this book we have presented a comprehensive framework that enables reuse of software artifacts. The fundamental assumption for this framework is that reuse is organised around precisely formulated and coherent requirements models. Such models are formulated in a unified Requirements Specification Language. Requirements specified in RSL form the basis for creating complete software cases which have the potential for reusing them. These software cases can be built using the standard UML notation where certain constraints and well-formedness rules should be applied. With appropriate guidelines for formulating architectural and detailed design models in place, we can define transformations that can translate the requirements specifications into design models. These transformations assure that software cases form coherently mapped units of reuse.

Having uniformly defined software cases we obtain the potential to organise a reuse-oriented software development process. Within this process, software cases are built from requirements into code using automatic transformations and other design activities performed by software developers. In this process, there are maintained clear traces leading from individual requirements to design solutions and then to code. These traces allow for finding proper solutions to problems formulated through requirements. Traceable software cases are stored in a reuse repository. When a new problem arises, the new requirements specification can be compared with the stored ones. After finding suitable old cases, their solutions can be adapted to the current problem. This adaptation is feasible due to existence of traces which point to design components and code elements that potentially need to be reworked.

The reuse framework presented in this book promises high levels of reuse. This could be possible both for completely new projects as well as for efforts in maintaining and extending the existing software systems. In the first case, a search engine could be used to find similar cases after formulating the initial requirements specification. This search could be done in local repositories of the given software development organisation. However, there is also possible a

vision of globally available software cases or partial software cases. With such global repositories, a "software case reuse market" could emerge.

For projects developing new versions of existing software there are two possibilities. First, the retrieval engine can precisely find places where the old version of software should be modified or extended. This is because by comparing the old and new requirements, the engine can trace differences into design and code. This gives software developers instant information on the size of the modification effort and reduces the time they would need to find appropriate elements to modify, by hand. The second possibility is to find solutions to newly introduced requirements in other stored software cases. It is often that the new functionality has already been implemented in other projects and can be easily retrieved and merged into the current workspace.

In order for the presented framework to become possible, a reuse engine has to be developed. Such an integrated tool is crucial, as the currently available tools do not support such a comprehensive reuse originating in requirements. Appropriate technologies, presented in the state of the art section, exist but are not properly linked and validated for the purpose. For instance, there exist commercial requirements engineering tools but they mainly operate on requirements as such, without necessary handling of the requirements representations. Some of the tools allow for linking requirements (or generally, any text) through hyperlinks but no links to other (especially - visual) software development artifacts are possible. We also have several tools to handle model transformations. These in turn seem to ignore requirements as suitable for transformation. Finally, retrieval technologies were used to retrieve software artifacts, but this was mainly done for individual models or artifacts.

In this book we have shown how to combine all the above technologies to create a comprehensive software reuse engine. There were determined necessary user requirements for such a tool, and its important architectural elements were identified. Within this architectural framework, appropriate technologies were discussed in more detail. It was shown that appropriate elements of the reuse engine can be based on the presented technologies, in order for the postulated functionality to become possible. All these technologies can be incorporated into a single reuse engine thanks to a unified meta-model for all necessary elements, as presented in this book. This meta-model covers the language for specifying queryable software cases (including the language for specifying requirements, and the language to specify model transformations) and the query language. Defining this coherent meta-model is an important prerequisite to build an appropriate tool to be used by software developers.

It has to be stressed that although the applied technologies form a complex technological framework, the resulting reuse engine can be easy to use by the developers. It is important to assure that the efforts to formulate reusable cases and then retrieve them are reduced to minimum. With the system presented in this book, no special effort is needed to transform a "regular" software case into a reusable one. All the information needed to retrieve such a case is already present in the requirements specification. This specification is struc-

tured in such a way that all functional and vocabulary aspects of the system can be easily compared with other such specifications. At the same time, requirements formulated in RSL can be automatically transformed into a query. This means that there is no special effort needed to formulate the query. All the query information is already incorporated into the newly created requirements specification. The developers need only to sketch some requirements in RSL and treat them as input to the query engine. The engine responds in a set of prioritised software cases relevant for the current problem.

The presented framework is currently the main subject of a EU funded project under the Information Society Technologies priority of the 6th Framework Programme for Scientific Research. The ReDSeeDS project [1] is divided into six major work packages. The first three consist in defining and validating the three technological elements as specified in Chapter 4. The fourth package is devoted to building the reuse engine. In the fifth package, the engine and the overall approach shall be validated in real life industry projects. Finally, the last package shall formulate a detailed methodology (process) as it was sketched in Section 2.2.

The presented approach to formulating software cases has already been validated within the ReDSeeDS project. The RSL was applied to several industry projects and validation results were formulated by the project Partners. Moreover a case study has been conducted where a complete software case has been developed, including the resulting code. Prior to this, several student projects were conducted as part of software engineering courses at the Warsaw University of Technology (see also [206]) and the University of Carlos III in Madrid.

The main elements found in Chapter 3 were presented to the students along with presenting the UML. Six different problem domains were chosen: fitness club, theme park, video and car rental, parking network and university campus. Altogether, 16 teams of 3-5 master-level students have prepared full software cases (except for code) containing a User Requirements Document (URD) and a Software Requirements Document (SRD). Some of the groups have also continued work by preparing an Architectural Design Document (ADD) and a Detailed Design Document (DDD) as described throughout Chapter 3.

The students were taught syntax and semantics of the case specification language both during lectures (around 10 hours devoted purely to SVO sentences and domain vocabulary) and hands-on tutorials (6 document validation sessions). The resulting deliverables contained between 34 and 46 use cases with short natural language descriptions and from 25 to 37 domain elements

[1] www.redseeds.eu, contract no. IST-2006-33596, coordinated by Infovide-Matrix, Poland with technical lead of Warsaw University of Technology and with University of Koblenz-Landau, Vienna University of Technology, Fraunhofer IESE, University of Latvia, HITeC e.V. c/o University of Hamburg, Heriot-Watt University, PRO DV, Cybersoft and Algoritmu Sistemos

briefly defined. Selected use cases (about 30-40% of the total) were described in detail with SVO scenarios. The groups were free to choose between textual and activity notation. While writing SVO sentences, the verbs were associated with appropriate nouns and included in domain elements definitions (treated as phrases). Some of the groups have used classical class diagrams for the domain vocabulary. Typically, the groups have written around 20 main course scenarios with some alternative ones. Around 40 phrases used within these scenarios were defined.

The groups that have continued their work in the next semester have transformed their requirements specifications into design documents. For most of the scenarios, interaction diagrams were created. Domain elements were transformed into design class diagrams, and also used within component diagrams (transformed into data transfer objects).

It has to be noted that no special tool was used to keep coherence of the SVO sentences and vocabulary phrases. This is similar to a situation within a typical requirements specification. However, despite a limited time to check the coherence, the students did manage to keep most of the hyperlinks valid (not broken). Only around 15-25% of the links had problems (usually minor inconsistencies in the names). Of course, their remarks show that keeping this coherence is time consuming, and a tool would dramatically help in this task. On the other hand, when transforming to design the students highly appreciated how the requirements specification was made internally consistent. This allowed to minimise efforts associated with ensuring consistency of interaction and component/class diagrams. The same observations were made when preparing the case study within ReDSeeDS.

The current book elaborates on several important issues associated with creating a comprehensive reuse framework. It has to be stressed that the presented approach is still far from being fully researched. Thus, appropriate further research and development activities are needed.

With the current state of efforts in defining the case specification languages and their meta-models, it is already possible to implement a tool fulfilling all of the presented requirements. An important issue here is to gain the possibility of validating various elements of the framework. Only the usage of a comprehensive tool can give us necessary information on applicability of the approach to real life projects. Thus the plans of further research (specifically within the ReDSeeDS project) assume using the created tool to validate the mechanisms for building and reusing software cases. It is very important to verify that it is possible to build a full transformation algorithm that would enable automatic translation of models and creation of coherently mapped cases. Moreover, a software case retrieval method has to be developed through validating several existing methods. Certainly, the transformation and query languages as presented in this book are not comprehensive enough and appropriate more elaborate languages and methods need to be used, as indicated in relevant sections.

Validation of the presented technologies in practice is an important basis for building a usable reuse platform. In the ReDSeeDS project this has been taken as an important element in planning of its schedule. The project assumes two iterations of research work. These two iterations are divided with the above mentioned validation work. Thus, after validating the developed platform (technologies with the tool), the results shall be used to improve the technologies and building the next version of the tool.

This final prototype tool could then be used in software development organisations to optimise their efforts by enabling easy access to gathered software knowledge. Having the tool spread more widely, a vision of software case reuse community could be fulfilled. Software producers could offer their software knowledge on the market or in public domain by publishing their software cases through publicly available interfaces. This could be an important factor in improving software manufacturing practices on a global market.

References

1. Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7(1):39–59, 1994.
2. Sushil Acharya and R Sadananda. Promoting software reuse using self organizing maps. *Neural Processing Letters*, 5(3):219–226, 1997.
3. Steve Adolph, Paul Bramble, Alistair Cockburn, and Andy Pols. *Patterns for Effective Use Cases*. Addison Wesley, 2002.
4. Ian Alexander. A taxonomy of stakeholders, human roles in system development. *International Journal of Technology and Human Interaction*, 1(1):23–59, 2005.
5. Ian Alexander and Neil Maiden, editors. *Scenarios, Stories, Use Cases*. John Wiley, 2004.
6. Klaus-Dieter Althoff, Andreas Birk, Susanne Hartkopf, Wolfgang Mller, Markus Nick, Dagmar Surmann, and Carsten Tautz. Systematic population, utilization, and maintenance of a repository for comprehensive reuse. *Lecture Notes in Computer Science*, 1756:25–50, 2000.
7. Scott W Ambler. A roadmap for Agile MDA. http://www.agilemodeling.com/essays/agileMDA.htm, 2005.
8. Albert Ambroziewicz. Software model library with visual query language. In *International Workshop on Model Reuse Strategies - MoRSe*, pages 33–36, 2006.
9. Albert Ambroziewicz and Ireneusz Bulwarski. Wyszukiwanie w bibliotece modeli oprogramowania za pomocą języka wizualnych zapytań (Searching in a software model library with a visual query language). Master's thesis, Warsaw University of Technology, 2005. supervised by Michał Śmiałek.
10. AONIX. *Using ACD Templates, Tool Documentation*, 2004.
11. Frank Armour and Granville Miller. *Advanced Use Case Modeling: Software Systems*. Addison-Wesley, 2000.
12. S Arnott and G Hinks. HMRC merger tops government IT spending. *Accountancy Age*, 4 Aug. 2005.
13. Hernán Astudillo, Gonzalo Génova, Michał Śmiałek, Juan Llorens Morillo, Pierre Metz, and Rubén Prieto-Diáz. Use cases in model-driven software engineering. *Lecture Notes in Computer Science*, 3844:262–271, 2006.
14. C Atkinson, J Bayer, C Bunse, E Kamsties, O Laitenberger, R Laqua, D Muthig, B Paech, J Wüst, and J Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.

15. F Baader, D Calvanese, D McGuinness, D Nardi, and P Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
16. Sidney C Bailin. Software reuse as ontology negotiation. *Lecture Notes in Computer Science*, 3107:242–253, June 2004.
17. Victor Basili and H Dieter Rombach. Support for comprehensive reuse. *IEEE Software Engineering Journal*, 6(5):303–316, 1991.
18. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
19. Alex E Bell. Death by UML fever. *Queue*, 2(1):72–80, March 2004.
20. Alex E Bell. UML fever: Diagnosis and recovery. *Queue*, 3(2):48–56, March 2005.
21. Birol Berkem. How to increase your business reactivity with UML/MDA. *Journal of Object Technology*, 2(6):117–138, Nov–Dec 2003.
22. J Bezivin, G Dupe, F Jouault, G Pitette, and J Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.
23. Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proc. 16th Annual International Conference on Automated Software Engineering ASE 2001*, pages 273–280, November 2001.
24. Robert Biddle, James Noble, and Ewan Tempero. From essential use cases to objects. In L Constantine, editor, *forUSE 2002 Proceedings*. Ampersand Press, 2002.
25. Robert Biddle, James Noble, and Ewan Tempero. Supporting reusable use cases. *Lecture Notes in Computer Science*, 2319:210–226, 2002.
26. Ted J Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5:169–225, 1998.
27. Solveig Bjornestad. Analogical reasoning for reuse of object-oriented specifications. *Lecture Notes in Computer Science*, 2689:50–64, 2003.
28. Maurits C Blok and Jacob L Cybulski. Reusing UML specifications in a constrained application domain. In *Proceedings of 1998 Asia Pacific Software Engineering Conference*, pages 196–202, 1998.
29. G Bockle, P Clements, J D McGregor, D Muthig, and K Schmid. Calculating ROI for software product lines. *IEEE Software*, 21(3):23–31, 2004.
30. Grady Booch. *Object-Oriented Design with Applications*. Benjamin–Cummings, Menlo Park, 1991.
31. Peter Braun and Frank Marschall. Transforming object oriented models with BOTL. *Electronic Notes in Theoretical Computer Science*, 72(3):1–15, 2003.
32. J M Carroll, editor. *Scenario-based design: Envisioning work and technology in system development*. Wiley, New York, 1995.
33. Rick Cattell. *The Object Database Standard: ODMG 2.0*. Morgan Kaufman, 1997.
34. Christine W Chan. Knowledge and software modeling using UML. *Software and Systems Modeling*, 3(4):294–302, 2004.
35. Ruzanna Chitchyan, Awais Rashid, Paul Rayson, and Robert Waters. Semantics-based composition for aspect-oriented requirements engineering. In *Proc. 6th International Conference on Aspect-Oriented Software Development (AOSD 2007)*, 2007.
36. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

37. Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice Hall, Englewood Cliffs, 1990.
38. Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice Hall, Englewood Cliffs, 1991.
39. Alistair Cockburn. Structuring use cases with goals. *Journal of Object-Oriented Programming*, 5(10):56–62, 1997.
40. Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.
41. Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2002.
42. Mike Cohn. *User Stories Applied*. Addison-Wesley, 2004.
43. Jim Conallen. *Building Web applications with UML*. Addison-Wesley Longman, Boston, MA, USA, 1999.
44. S Condon, C Seaman, V Basili, S Kraft, J Kontio, and Y Kim. Evolving the reuse process at the Flight Dynamics Division (FDD) Goddard Space Flight Center. In *Proceedings of the 21st Annual Software Engineering Workshop, NASA/SEL*, 1996.
45. Larry L Constantine. What do users want? Engineering usability into software. *Windows Tech Journal*, 1995. revised in 2000, http://www.foruse.com/articles/whatusers.htm.
46. Larry L Constantine and Lucy A D Lockwood. Structure and style in use cases for user interface design. In M van Harmelen, editor, *Object-Modeling and User Interface Design*. Addison-Wesley, 2001.
47. James R Cordy. TXL - a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31, 2004.
48. Jacob L Cybulski, Ralph D (Butch) Neal, Anthony Kram, and Jeffrey C Allen. Reuse of early life-cycle artifacts: workproducts, methods and tools. *Annals of Software Engineering*, 5(0):227–251, 1998.
49. Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Nonfunctional requirements: from elicitation to conceptual models. *IEEE Transactions on Software Engineering*, 30(5):328–350, May 2004.
50. Krzysztof Czarnecki, , and Ulrich W Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.
51. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
52. Benedicte Dano, Henri Briand, and Franck Barbier. An approach based on the concept of use case to produce dynamic object-oriented specifications. In *3rd IEEE International Symposium on Requirements Engineering (RE'97)*, pages 54–64. IEEE Computer Society, 1997.
53. E S de Almeida, A Alvaro, D Lucredio, V C Garcia, and S R de Lemos Meira. A survey on software reuse processes. In *IEEE International Conference on Information Reuse and Integration*, pages 66–71, 2005.
54. P Devanbu, P G Brachman, R J andSelfridge, and B W Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, 1991.
55. Dragan Djuric, Dragan Gasevic, and Vladan Devedzic. Ontology modeling and MDA. *Journal of Object Technology*, 4(1):109–128, 2005.
56. K Duddy, A Gerber, M J Lawley, K Raymond, and Steel J. Model transformation: A declarative, reusable patterns approach. In *Proc. 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2003*, pages 174–195, Brisbane, Australia, September 2003.

57. A Duran, A Ruiz-Cortes, R Corchuelo, and M Toro. Supporting requirements verification using XSLT. In *Proc. IEEE Joint International Conference on Requirements Engineering (RE'02)*, pages 165–172, 2002.
58. Gerald Ebner and Hermann Kaindl. Tracing all around in reengineering. *IEEE Software*, 19(3):70–77, 2002.
59. Alexander Egyed. A scenario-driven approach to traceability. In *ICSE '01: Proc. 23rd International Conference on Software Engineering*, pages 123–132, Washington, DC, USA, 2001. IEEE Computer Society.
60. Alexander Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, 2003.
61. European Space Agency. *ESA PSS-05-0 Software Engineering Standards, Issue 2, Revision 1*, 1994.
62. Joerg Evermann and Yair Wand. Toward formalizing domain modeling semantics in language syntax. *IEEE Transactions on Software Engineering*, 31(1):21–37, January 2005.
63. John Favaro. A comparison of approaches to reuse investment analysis. In *Proc. 4th International Conference on Software Reuse*, pages 136–145, 1996.
64. Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
65. Donald Firesmith and Brian Henderson-Sellers. *The OPEN Process Framework. An Introduction*. Addison-Wesley, 2001.
66. Gilles Fouqué and Stan Matwin. A case-based approach to software reuse. *Journal of Intelligent Information Systems*, 2(2):165–197, 1993.
67. Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1996.
68. Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison-Wesley, 2004.
69. William B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, July 2005.
70. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.
71. David Gelperin. Precise use cases. Technical report, LiveSpecs Software, 2004.
72. Gonzalo Génova, Juan Llorens, Pierre Metz, Rubén Prieto-Diaz, and Hernán Astudillo. Open issues in industrial use case modeling. *Lecture Notes in Computer Science*, 3297:52–61, 2005.
73. A Gerber, M J Lawley, K Raymond, J Steel, and A Wood. Transformation: The missing link of MDA. *Lecture Notes in Computer Science*, 2505:90–105, 2002.
74. Martin Glinz. A lightweight approach to consistency of scenarios and class models. In *Proc. 4th International Conference on Requirements Engineering (ICRE'00)*, pages 49–58, 2000.
75. H Gomaa, L Kerschberg, V Sugumaran, C Bosch, I Tavakoli, and L O'Hara. A knowledge-based software engineering environment for reusable software requirements and architectures. *Automated Software Engineering*, 3(3–4):285–307, 1996.
76. Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley, 2004.
77. Paulo Gomes. Software design retrieval using bayesian networks and WordNet. *Lecture Notes in Computer Science*, 3155:184–197, 2004.

78. Paulo Gomes, Francisco C Pereira, Paulo Carreiro, Paulo Paiva, Nuno Seco, José Luís Ferreira, and Carlos Bento. Solution verification in software design: A CBR approach. *Lecture Notes in Computer Science*, 2689:171–185, 2003.
79. Paulo Gomes, Francisco C Pereira, Paulo Carreiro, Paulo Paiva, Nuno Seco, José Luís Ferreira, and Carlos Bento. Case-based adaptation for UML diagram reuse. *Lecture Notes in Computer Science*, 3215:678–686, 2004.
80. Markus Grabert and Derek G Bridge. Case-based reuse of software examplets. *Journal of Universal Computer Science*, 9(7):627–641, 2003.
81. Ian M Graham. *Migrating to Object Technology.* Addison-Wesley, Workingham, 1994.
82. Ian M Graham. Task scripts, use cases and scenarios in object-oriented analysis. *Object-Oriented Systems*, 3(3):123–142, 1996.
83. Ian M Graham. *Object-Oriented Methods Principles & Practice.* Pearson Education, 2001.
84. Jack Greenfield and Keith Short. *Software Factories. Assembling Applications with Patterns, Models, Frameworks and Tools.* Wiley, Indianapolis, Indiana, 2004.
85. Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: RML revisited. In *ICSE '94: Proc. 16th International Conference on Software Engineering*, pages 135–147, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
86. Sol J Greenspan, Alexander Borgida, and John Mylopoulos. A requirements modeling language and its logic. *Information Systems*, 11(1):9–23, 1986.
87. Paweł Gryczon and Piotr Stańczuk. Obiektowy system konstrukcji scenariuszy przypadków uzycia (Object-oriented use case scenario construction system). Master's thesis, Warsaw University of Technology, 2002.
88. Haitham Hamza and Mohamed E Fayad. Applying analysis patterns through analogy: Problems and solutions. *Jornal of Object Technology*, 3(4):197–208, April 2004.
89. M Hause and F Thom. Modeling high level requirements in uml/sysml. In *INCOSE 2005 UK Spring Conference*, pages 10–15, 2005.
90. Brian Henderson-Sellers and Julian Edwards. *Booktwo of Object-Oriented Knowledge.* Prentice Hall, Englewood Cliffs, 1994.
91. Brian Henderson-Sellers, Anthony J H Simons, and Houman Younessi. *The OPEN Toolbox of Techniques.* Addison-Wesley Longman, Wokingham, 1998.
92. Scott Henninger. Case-based knowledge management tools for software development. *Automated Software Engineering*, 4(1):319–340, 1997.
93. L Hotz, K Wolter, T Krebs, S Deelstra, M Sinnema, J Nijhuis, and J MacGregor. *Configuration in Industrial Product Families - The ConIPF Methodology.* AKA Verlag, 2005.
94. Russell R Hurlbut. A survey of approaches for describing and formalizing use cases. Technical Report XPT-TR-97-03, Expertech Ltd., 1997.
95. Russell R Hurlbut. *Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models.* PhD thesis, Illinois Institute of Technology, 1998.
96. International Standards Organization. *Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2*, 2005. ISO/IEC 19501:2005.

97. I Jacobson, M Christerson, P Jonsson, and G Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, 1992.
98. Young Jun Kim and Chung Tae Kim. Design of object model reuse system by CBR in system analysis. *International Journal of Software Engineering and Knowledge Engineering*, 14(3):277–290, 2004.
99. Hermann Kaindl. Using hypertext for semiformal representation in requirements engineering practice. *The New Review of Hypermedia and Multimedia*, 2:149–173, 1996.
100. Hermann Kaindl. Difficulties in the transition from OO analysis to design. *IEEE Software*, 16(5):94–102, Sept./Oct. 1999.
101. Hermann Kaindl. A scenario-based approach for requirements engineering: Experience in a telecommunication software development project. *Systems Engineering*, 8(3):197–209, 2005.
102. Hermann Kaindl and John Mylopoulos. Why is it so difficult to introduce requirements engineering research results into mainstream requirements engineering practice? In *Proceedings of the Twelfth Conference on Advanced Information Systems Engineering (CAiSE 2000)*, pages 7–12, Stockholm, Sweden, June 2000. Springer-Verlag, Heidelberg, Germany. Panelists: S. Brinkkemper, J. A. Bubenko, B. Farbey and I. Jacobson.
103. Hermann Kaindl, Michał Śmiałek, Davor Svetinovic, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Hannes Schwarz, Daniel Bildhauer, John P Brogan, Kizito Ssamula Mukasa, Katharina Wolter, and Thorsten Krebs. Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project, 2007. www.redseeds.eu.
104. Audris Kalnins, Janis Barzdins, and Edgars Celms. Basics of model transformation language MOLA. In *Workshop on Model Driven Development (WMDD 2004)*, 2004.
105. Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language MOLA. *Lecture Notes in Computer Science*, 3599:14–28, 2004.
106. Audris Kalnins, Agris Sostaks, Edgars Celms, Elina Kalnina, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Volker Riediger, Hannes Schwarz, Daniel Bildhauer, Sevan Kavaldjian, Roman Popp, and Jurgen Falb. Reuse-oriented modelling and transformation language definition. Project Deliverable D3.2, ReDSeeDS Project, 2007. www.redseeds.eu.
107. K C Kang, J Lee, and P Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.
108. Kyo C Kang, Sholom Cohen, Robert Holibaugh, James Perry, and A Spencer Peterson. A reuse-based software development methodology. Technical report, Software Engineering Institute, Carnegie Mellon University, 1992. Special Report CMU/SEI-92-SR-4.
109. Even-André Karlsson. *Software Reuse: A Holistic Approach.* John Wiley, 1991.
110. Panagiotis Katalagarianos and Yannis Vassiliou. On the reuse of software: A case-based approach employing a repository. *Automated Software Engineering*, 2(1):55–86, 1995.
111. Anneke G Kleppe, Jos B Warmer, and Bast Wim. *MDA Explained, The Model Driven Architecture: Practice and Promise.* Addison-Wesley, Boston, 2003.
112. J Koehler, R Hauser, S Kapoor, F Y Wu, and S Kumaran. A model-driven transformation method. In *Proc. 7th IEEE International Enterprise Dis-*

tributed Object Computing Conference EDOC 2003, pages 186–197, September 2003.
113. A Koenigs and A Schuerr. Multi-domain integration with mof and extended triple graph grammars. In J Bezivin and R Heckel, editors, *Language Engineering for Model-Driven Software Development, Dagstuhl Seminar Proceedings, IBFI*, 2005.
114. P Kruchten. Agility with the RUP. *Rational Edge*, 3:11p, 2002.
115. Philippe Kruchten. *The Rational Unified Process: An Introduction, 3rd ed.* Addison Wesley, 2003.
116. Miguel A Laguna, Oscar López, and Yania Crespo. Reuse, standardization, and transformation of requirements. *Lecture Notes in Computer Science*, 3107:329–338, 2004.
117. Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Use Case Approach, Second Edition.* Addison-Wesley, 2003.
118. N G Lester, F G Wilkie, and D W Bustard. Applying UML extensions to facilitate software reuse. *Lecture Notes in Computer Science*, 1618:393–405, 1999.
119. Tihamer Levendovszky, Gabor Karsai, Miklos Maroti, Akos Ledeczi, and Hassan Charaf. Model reuse with metamodel-based transformations. *Lecture Notes in Computer Science*, 2319:166–179, Jan 2002.
120. Juan Llorens, José M Fuentes, Rubén Prieto-Diaz, and Hernán Astudillo. Incremental software reuse. *Lecture Notes in Computer Science*, 4039:386–389, 2006.
121. Chung-Horng Lung, Joseph E Urban, and Gerald T Mackulak. Analogy-based domain analysis approach to software reuse. *Requirements Engineering*, 11(4):22p, 2006.
122. Paul Luttikhuizen, editor. *Requirements Modelling and Traceability.* ESAPS project, ITEA 99005, 2001. Document no. Philips-WP3-0106-01.
123. Robert A Maksimchuk and Eric J Naiburg. *UML for Mere Mortals.* Addison-Wesley, 2005.
124. Mike Mannion, Hermann Kaindl, and Joe Wheadon. Reusing single system requirements from application family requirements. In *Proc. International Conference on Software Engineering 1999 (ICSE'99)*, 1999.
125. M D McIlroy. Mass produced software components. In P Naur, B Randell, and J N Buxton, editors, *Software engineering concepts and techniques, Proceedings of NATO Conference on Software Engineering*, pages 88–98, New York, 1969.
126. Nenad Medvidovic, Paul Gruenbacher, Alexander Egyed, and Barry W Boehm. Software model connectors: Bridging models across the software lifecycle. In *Proc. 13th Int. Conference on Software Engineering and Knowledge Engineering (SEKE'01)*, pages 387–396, 2001.
127. Stephen J Mellor and Marc J Balcer. *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley, 2002.
128. Stephen J Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model Driven Architecture.* Addison-Wesley, 2004.
129. T Mens and P van Gorp. A taxonomy of model transformation. In *Proc. Int'l Workshop on Graph and Model Transformation, GraMoT 2005*, 2005.
130. T Menzies and J S Di Stefano. More success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 29(5):474–477, 2003.

131. P Metz, J O'Brien, and W Weber. Specifying use case interaction: Types of alternative courses. *Journal of Object Technology*, 2(2):111–131, March-April 2003.
132. Pierre Metz, John O'Brien, and Wolfgang Weber. Against use case interleaving. *Lecture Notes in Computer Science*, 2185:472–486, 2001.
133. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.
134. Amir Michail. Data mining library reuse patterns using generalized association rules. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 167–176, New York, NY, USA, 2000. ACM Press.
135. A Mili, S F Chmiel, R Gottumukkala, and L Zhang. An integrated cost model for software reuse. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 157–166, 2000.
136. A Mili, R Mili, and R T Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5:349–414, 1998.
137. Granville Miller. *The Microsoft Solutions Framework for Agile Software Development: The Definitive Guide to Microsoft's New Agile Methodology.* Addison-Wesley, 2007. in press.
138. Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide Version 1.0.1, omg/03-06-01.* Object Management Group, 2003.
139. Maurizio Morisio, Michel Ezran, and Colin Tully. Success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 28(4):340–357, 2002.
140. John G Morris and Christine M Mitchell. A case-based support system to facilitate software reuse. In *1997 IEEE International Conference on Systems, Man, and Cybernetics 'Computational Cybernetics and Simulation'*, volume 1, pages 232–237, 1997.
141. Ezra K Mugisa. A reuse triplet view of UML. In *Proceedings of 2003 IEEE SoutheastConf.*, pages 126–133, 2003.
142. Jerzy R Nawrocki, Michał Jasiński, Bartosz Walter, and Adam Wojciechowski. Extreme programming modified: Embrace requirements engineering practices. In *Proc. 10th IEEE Joint International Conference on Requirements Engineering (RE 2002)*, pages 303–310, Essen, Germany, 2002. IEEE Computer Society.
143. Jerzy R Nawrocki, Tomasz Nędza, Mirosław Ochodek, and Łukasz Olek. Describing business processes with use cases. In *Proc. 9th International Conference on Business Information Systems (BIS 2006)*, volume 85 of *Lecture Notes in Informatics*, pages 13–27, Klagenfurt, Austria, 2006. GI.
144. Jerzy R Nawrocki and Łukasz Olek. UC Workbench - a tool for writing use cases and generating mockups. *Lecture Notes in Computer Science*, 3556:230–234, 2005.
145. J M Nerson. Applying object-oriented analysis and design. *Communications of the ACM*, 35(9):63–74, 1992.
146. Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE - Future of SE Track*, pages 35–46, 2000.
147. Object Management Group. *MOF 2.0 Query / Views / Transformations RFP, ad/2002-04-10*, 2002.
148. Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification, Final Adopted Specification, ptc/03-10-04*, 2003.
149. Object Management Group. *Reusable Asset Specification: Final Adopted Specification, ptc/04-06-06*, 2004.

150. Object Management Group. *UML Profile for Patterns Specification, version 1.0, formal/04-02-04*, 2004.
151. Object Management Group. *Unified Modeling Language: Superstructure, version 2.0, Revised Final Adopted Specification, ptc/04-10-02*, 2004.
152. Object Management Group. *OCL 2.0, Final Specification, ptc/05-06-06*, 2005.
153. Object Management Group. *Unified Modeling Language: Infrastructure, version 2.0, formal/05-07-05*, 2005.
154. Object Management Group. *Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04*, 2005.
155. Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, 2006.
156. Ken Orr and Randy Hester. Beyond MDA 1.0: executable business processes from concept to code. *MDA Journal*, 11:2–7, 2004.
157. Gunnard Overgaard and Karin Palmkvist. *Use Cases: Patterns and Blueprints.* Addison Wesley, 2005.
158. Stephen R Palmer and John M Felsing. *A Practical Guide to Feature-Driven Development.* Prentice Hall PTR, 2002.
159. Ying Pan, Lei Wang, Lu Zhang, Bing Xie, and Fuqing Yang. Relevancy based semantic interoperation of reuse repositories. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 211–220, New York, NY, USA, 2004. ACM Press.
160. Tom Pender. *UML Bible.* John Wiley & Sons, New York, NY, USA, 2003.
161. K Periyasamy and J Chidambaram. A method for structural compatibility in software reuse using requirements specification. In *21st Annual International Computer Software and Applications Conference COMPSAC '97*, pages 426–433, 1997.
162. James Petro, Michael E Fotta, and David B Weisman. Model-based reuse repositories-concepts and experience. In *Proceedings of Seventh International Workshop on Computer-Aided Software Engineering*, pages 60–69, 1995.
163. Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell.* O'Reilly, 2005.
164. J Poulin. *Measuring Software Reuse.* Addison-Wesley, 1997.
165. Rubén Prieto-Diaz. Domain analysis: an introduction. *SIGSOFT Softw. Eng. Notes*, 15(2):47–54, 1990.
166. QVT–Merge Group, OMG. *Revised submission for MOF 2.0 Query / View / Transformation RFP*, 2005. ad/2005-03-02P.
167. QVT-Partners. *Revised submission for MOF 2.0 Query / Views / Transformations RFP*, 2003.
168. Chris Raistrick, Paul Francis, and John Wright. *Model Driven Architecture with Executable UML.* Cambridge University Press, 2004.
169. Volker Riediger, Daniel Bildhauer, Hannes Schwarz, Audris Kalnins, Agris Sostaks, Edgars Celms, Michał Śmiałek, and Albert Ambroziewicz. Software case meta-model definition. Project Deliverable D3.1, ReDSeeDS Project, 2007. www.redseeds.eu.
170. W N Robinson and H G Woo. Finding reusable UML sequence diagrams automatically. *IEEE Software*, 21(5):60–67, 2004.
171. C Rolland and C Ben Achour. Guiding the construction of textual use case specifications. *Data & Knowledge Engineering*, 25(1–2):125–160, March 1998.
172. C Rolland, C Souveyet, and C. Ben Achour. Guiding goal modeling using scenarios. *IEEE Transactions on Software Engineering*, 24(12):1055–1071, 1998.

173. Pascal Roques. *UML in Practice: The Art of Modeling Software Systems Demonstrated through Worked Examples and Solutions*. John Wiley & Sons, 2004.
174. Dough Rosenberg and Kendall Scott. *Use Case Driven Object Modeling with UML*. Addison Wesley, 1999.
175. Dough Rosenberg, Matt Stephens, and Mark Collins-Cope. *Agile Development with ICONIX Process: People, Process, and Pragmatism*. APress, 2005.
176. Marcus A Rothenberger, Kevin J Dooley, Uday R Kulkarni, and Nader Nada. Strategies for software reuse: a principal component analysis of reuse practices. *IEEE Transactions on Software Engineering*, 29(9):825–837, September 2003.
177. J Rumbaugh, M Blaha, W Pemerlani, F Eddy, and W Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.
178. J Rumbaugh, I Jacobson, and G Booch. *The Unified Modelling Language Reference Guide*. Addison-Wesley, Menlo Park, 1999.
179. Motoshi Saeki. Patterns and aspects for use cases: reuse techniques for use case descriptions. In *4th International Conference on Requirements Engineering*, page 62, 2000.
180. M Sasikumar. Case-based reasoning for software reuse. In *Knowledge Based Computer Systems - Research and Applications (International Conference on Knowledge-Based Computer Systems)*, pages 31–42, London, 1996. Narosa Publishing House.
181. Douglas C Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report*, 11(1), January 1999.
182. Geri Schneider and Jason P Winters. *Applying Use Cases: A Practical Guide, Second Edition*. Addison-Wesley, 2001.
183. A Schuerr. PROGRES: A VHL-language based on graph grammars. *Lecture Notes in Computer Science*, 532:641–659, 1991.
184. SEI. *Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995. Software Engineering Institute, Carnegie Mellon University.
185. Richard W Selby. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, 31(6):495–510, June 2005.
186. Anthony J H Simons. Use cases considered harmful. In *Proceedings of the 29th Conference on Technology of Object-Oriented Languages and Systems-TOOLS Europe'99*, pages 194–203, Nancy, France, June 1999. IEEE Computer Society Press.
187. Michał Śmiałek. Discovery - third generation OOA&D method: Case study with proposition of notation and process. Master's thesis, University of Sheffield, Department of Computer Science, 1996.
188. Michał Śmiałek. Global reuse strategy based on use cases. In *OOPSLA 2000 Companion, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 49–50, Minneapolis, 2000.
189. Michał Śmiałek. Accommodating informality with necessary precision in use case scenarios. *Journal of Object Technology*, 4(6):59–67, August 2005.
190. Michał Śmiałek. Can use cases drive software factories? In *Workshop on Use Cases in Model-Driven Software Engineering (WUsCaM)*, 2005.
191. Michał Śmiałek. From user stories to code in one day? *Lecture Notes in Computer Science*, 3556:38–47, 2005.

192. Michał Śmiałek. Profile suite for model transformations on the computation independent level. *Lecture Notes in Computer Science*, 3297:264–268, 2005.
193. Michał Śmiałek. *Zrozumieć UML 2.0. Metody modelowania obiektowego*. Helion, 2005.
194. Michał Śmiałek. Mechanisms for requirements based model reuse. In *International Workshop on Model Reuse Strategies - MoRSe*, pages 17–20, 2006.
195. Michał Śmiałek and Łukasz Balcerek. Knowledge-based content management system application design methodology. Technical report, ICONS Project, 2003. Document D25.
196. Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, and Tomasz Straszak. Scenario construction tool based on extended UML metamodel. *Lecture Notes in Computer Science*, 3713:414–429, 2005.
197. Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, and Tomasz Straszak. Writing coherent user stories with tool support. *Lecture Notes in Computer Science*, 3556:247–250, 2005.
198. Michał Śmiałek and Andrzej Kardas. Model management based on a visual transformation language. In Krzysztof Zieliński and Tomasz Szmuc, editors, *Software engineering - evolution and emerging technologies*, pages 341–352. IOS Press, 2005.
199. Michał Śmiałek, Markus Nick, Audris Kalnins, Juergen Falb, and Rob Pooley, editors. *International Workshop on Model Reuse Strategies - MoRSe*. Fraunhofer IRB, 2006.
200. Stephane S Somé. Beyond scenarios: Generating state models from use cases. In *Scenarios and state machines: models, algorithms and tools - ICSE 2002 Workshop*, Orlando. Florida, 2002.
201. Andreas Speck, Elke Pulvermller, Ragnhild Van Der Straeten, Ralf H Reussner, and Matthias Clauss. Model-based software reuse. *Lecture Notes in Computer Science*, 2548:135–146, 2002.
202. Kazimierz Subieta. *Teoria i konstrukcja obiektowych języków zapytań (Theory and construction of object query languages)*. Wydawnictwo PJWSTK, 2004.
203. A G Sutcliffe, N A M Maiden, S Minocha, and D Manuel. Supporting scenario–based requirements engineering. *IEEE Transactions on Software Engineering*, 24(12):1072–1088, December 1998.
204. Alistair Sutcliffe and Neil Maiden. The domain theory for requirements engineering. *IEEE Transactions on Software Engineering*, 24(3):174–196, 1998.
205. SysML Partners. *Systems Modeling Language (SysML) Specification, version 0.9*, 2005.
206. Robert Szmurło and Michał Śmiałek. Teaching software modeling in a simulated project environment. *Lecture Notes in Computer Science*, 4364:301–310, 2007.
207. G Taentzer. AGG: A graph transformation environment for modeling and validation of software. *Lecture Notes in Computer Science*, 3062:446–453, 2004.
208. Carsten Tautz and Klaus-Dieter Althoff. Using case-based reasoning for reusing software knowledge. *Lecture Notes in Computer Science*, 1266:156–165, 1997.
209. Dave Thomas. MDA: Revenge of the modelers or UML utopia? *IEEE Software*, 21(3):22–24, 2004.
210. Juha-Pekka Tolvanen. Domain-specific modeling: No one size fits all. *Lecture Notes in Computer Science*, 3713:279, 2005.
211. University of Paderborn. *Fujaba: From UML to Java and back again*, 2005. http://www.fujaba.de/.

212. Klaas G van den Berg and Anthony J H Simons. Control flow semantics of use cases in UML. *Information and Software Technology*, 41(10):651–659, 1999.
213. A van Deursen, J Heering, and P Klint, editors. *Algebraic Specification*. ACM Press, 1996.
214. Han van Loon. *Process Assessment and ISO/IEC 15504 Reference Book*. Kluwer International Series in Engineering & Computer Science. Springer, 2004.
215. J van Wijngaarden and E Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, 2003. technical report.
216. E Visser. Stratego: a language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357–361, 2001.
217. E Visser. Dagstuhl workshop on trees vs. graphs. In *Transformation Techniques in Software Engineering*, 2005. Dagstuhl Seminar No. 05161.
218. W3C Web Services Architecture Working Group. *XSL Transformations (XSLT), Version 1.0*, 1999. W3C Recommendation.
219. K Watahiki and M Saeki. Scenario patterns based on case grammar approach. In *5th IEEE International Symposium on Requirements Engineering*, pages 300–301, 2001.
220. K Weidenhaupt, K Pohl, M Jarke, and P Haumer. Scenarios in system development: current practice. *IEEE Software*, 15(2):34–45, 1998.
221. D Weiss and C T R Lai. *Software Product-Line Engineering*. Addison-Wesley, 1999.
222. C A Welty and D A Ferrucci. A formal ontology for re-use of software architecture documents. In *14th IEEE International Conference on Automated Software Engineering*, pages 259–262, 1999.
223. Jon Whittle. Specifying precise use cases with use case charts. In *MoDELS'05 Workshop on Use Cases in Model Driven Software Engineering*, 2005.
224. Keith Williamson and Michael Healy. Deriving engineering software from requirements. *Journal of Intelligent Manufacturing*, 11(1):3–28, 2000.
225. R J Wirfs-Brock and R E Johnson. A survey on current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.
226. R J Wirfs-Brock and B Wilkerson. Object-orientred design: A responsibility driven approach. In *OOPSLA'89 Proceedings*, pages 71–75, 1989.
227. H G Woo and W N Robinson. Reuse of scenario specifications using an automated relational learner: a lightweight approach. In *Proceedings of IEEE Joint International Conference on Requirements Engineering*, pages 173–180, 2002.
228. Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 513–523, New York, NY, USA, 2002. ACM Press.
229. Chen Yonghao and Betty H C Cheng. Facilitating an automated approach to architecture-based software reuse. In *12th IEEE International Conference on Automated Software Engineering*, pages 238–245, 1997.
230. Albert Zündorf. Fujaba as a QVT language for MDA. In *Transformation Techniques in Software Engineering*, 2005. Dagstuhl Seminar No. 05161.

TWORZENIE OPROGRAMOWANIA PRZY POMOCY REUŻYWALNYCH PRZYPADKÓW OPARTYCH NA WYMAGANIACH

Streszczenie

Zdecydowana większość projektów konstrukcji oprogramowania wydaje się ignorować wiedzę na temat rozwiązanych wcześniej problemów. Można to wytłumaczyć trudnościami w ponownym wykorzystaniu wiedzy w tak złożonej dziedzinie jak inżynieria oprogramowania. Brakuje efektywnych mechanizmów znajdowania i ponownego wykorzystania rozwiązań minionych problemów, podobnych do stojących przed nami w danej chwili. Podstawową kwestią, której rozwiązanie jest celem tej książki jest powyższa niemożność „reużycia" wiedzy o rozwiązanych już problemach w konstrukcji oprogramowania. W tej książce zaproponowano proces systematycznego wykorzystywania tzw. przypadków programistycznych (ang. software case). Każdy przypadek programistyczny zawiera precyzyjnie sformułowane stwierdzenie problemu w formie modelu wymagań. Wszystkie elementy tego modelu mogą być przełożone na odpowiednie elementy rozwiązania sformułowanego problemu. To rozwiązanie jest złożone z precyzyjnie wyrażonych modeli projektowych oraz kodu. Przypadki programistyczne mogą być ponownie wykorzystane na podstawie ich podobieństwa do aktualnie tworzonego systemu (aktualnego przypadku programistycznego). To podobieństwo może być określone poprzez porównanie aktualnego (być może jeszcze niekompletnego) modelu wymagań z modelami wcześniej wytworzonych przypadków. Wcześniejsze rozwiązanie może być z łatwością ponownie wykorzystane poprzez jego modyfikację w miejscach oznaczonych jako wymagające przeróbek aby dostosować je do aktualnego problemu.

Książka zawiera szczegółową dyskusję kwestii, które umożliwiają skonstruowanie kompleksowego systemu ponownego wykorzystania opartego na wymaganiach. Opisane są mechanizmy i narzędzia wspomagające taki system. Przedstawiono wizję zorganizowania procesu ponownego wykorzystania, łącznie ze wskazówkami dla organizacji wytwarzających oprogramowanie. Oznacza to również wykorzystanie konkretnego, precyzyjnego języka specyfikacji wymagań i projektowania systemów. Proces i język są zdefiniowane zarówno formalnie jak i od strony praktycznej. Książka wprowadza konkretną składnię dla elementów przypadków programistycznych: wymagań, architektury i projektu szczegółowego. Ta składnia jest wykorzystana do formułowania ich w sposób systematyczny. Jednocześnie podano techniki dla transformacji modeli w celu utworzenia spójnej ścieżki od wymagań do kodu. Przedstawiono również określone mechanizmy porównywania i odszukiwania przypadków programistycznych. Dotyczy to również języka zapytań odpowiedniego dla formułowania kwerend pozwalających na dobieranie modeli wymagań, w ten sposób pozwalając na ponowne wykorzystanie rozwiązań wyspecyfikowanych wymaganiami.

Słowa kluczowe: ponowne użycie oprogramowania, wymagania oprogramowania, transformacje modeli oprogramowania, metodyki wytwarzania oprogramowania

Contents